VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vizualizace komplexních sítí pomocí 3D enginu Unity

Complex Network Layout Visualization in 3D Engine Unity

Daniel Milián

Bakalářská práce

Vedoucí práce: Ing. Martin Němec, Ph.D.

Ostrava, 2021

Abstrakt

Tato bakalářská práce se zabývá vizualizací komplexních sítí pomocí herního enginu Unity 3D. V teoretické části se práce zabývá cílem a možnostmi, teorii grafů, komplexními sítěmi, samotným Unity enginem a renderovací technikou využitou v aplikaci. Praktická část navazuje na teorií grafů a komplexních sítí, ve které je popsaná implementace takových grafů. Součásti práce je také popis implementace nástrojů pro práci s těmito grafy a optimalizace aplikace.

Klíčová slova

Unity 3D, Virtuální realita, Vizualizaci grafů

Abstract

This bachelor thesis deals with the visualization of complex networks using the Unity 3D game engine. The theoretical part deals with the goal and possibilities, graph theory, complex networks, the Unity engine itself and the rendering technique used in the application. The practical part follows the theory of graphs and complex networks, which describes the implementation of such graphs. Part of the work is also a description of the implementation of tools for working with these graphs and optimization of this application.

Keywords

Unity 3D, Virtual Reality, Graph visualization

Poděkování

Rád bych poděkoval panu Ing. Martinu Němcovi, Ph.D. za možné návrhy a nasměrování, také škole za půjčení zařízení HTC Vive, který byl velice užitečný při vývoji a testování aplikace.

Obsah

Seznam použitých zkratek a symbolů

VR	–	Virtual Reality
XR	–	Extended Reality
CPU	–	Central Processing Unit
GPU	–	Graphics Processing Unit
px	–	Pixel
URP	–	Universal Render Pipeline
HDRP	–	High Definition Render Pipeline
DOTS	–	Data-Oriented Technology Stack
ECS	–	Entity Component System
GDF	–	GUESS Graph Data formát
GEXF	–	Graph Exchange XML formát
CSV	–	Comma-separated values
UI	–	User Interface

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

V dnešní době není možné, aby jeden nebo více lidí bylo schopno analyzovat data efektivně bez jakýchkoliv pomocných prostředků. Existuje mnoho takových prostředků, jedna z nich je grafická vizualizace. Grafická vizualizace nám dovoluje vidět v datech výjimečné informace. Dnes jsme schopni data vizualizovat pomocí mnoha aplikací, bohužel jen jako dvoudimenzionální obrázek nebo plochu. Tato bakalářská práce se zabývá vizualizací těchto dat ve třídimenzionálním světě s možností realtime manipulace za pomocí herního enginu Unity 3D [1] a virtuální reality [2]. Unity 3D je v současné době jeden z nejvíce rozvíjejících se herních enginů na světě. Díky svému širokému poli působnosti se s ním můžeme setkat v mnohá hrách, ale taktéž i v jiných odvětvích. Díky jeho možnostem a flexibilitě ho lze efektivně využít i k jiným účelům, než vytváření her. Unity 3D již podporuje technologie virtuální reality, augmentované reality i mixované reality, zkráceně XR. Dnes na trhu není žádná aplikace zaměřená na vizualizaci grafu za pomocí virtuální reality, a proto cílem je vytvoření této aplikace, která je schopná vizualizovat komplexní grafy za pomocí virtuální reality, primárně pro zařízení HTC Vive [3] a Oculus Quest 2 [4]. Je pochopitelné, že každá nová aplikace funguje jinak oproti ostatním aplikacím a lidé si musí zvykat na novou nebo rozdílnou funkcionalitu, a proto je taktéž cílem dodat co nejvíce možností na které jsou lidé zvyklí. Zařízení HTC Vive a Oculus Quest 2 nativně běží ve frekvenci 90Hz, proto byla potřeba zajistit, aby i aplikace běžela plynule a bez záseků na této snímkové frekvenci. Z toho důvodu je jedna samostatná kapitola věnována i optimalizacím.

Na scéně se renderují jen ty nejvíce užitečné objekty pro práci, jako jsou uzly a hrany grafu, který jsme vytvořili za pomoci funkcí, které nám aplikace nabízí. Taktéž se na scéně nachází uživatelské rozhraní s kterým aktuálně pracujeme.

Kapitola 2

Cíl a možnosti

Cílem je vytvořit schopný nástroj pro pohodlnou a efektivní práci s komplexními grafy ve virtuální realitě. Na trhu existuje mnoho nástrojů k vizualizaci grafu, avšak málo z nich je schopných vizualizovat graf ve virtuální realitě. Jedna z těchto aplikací se jmenuje Virtualitics [5], bohužel však není dostupná všem uživatelům, ale jen po domluvě. Virtuální realita v dnešní době otevírá mnoho možností, proto je třeba těchto možností využít.

Pro tento nástroj byl vybrán herní engine Unity oproti Unreal Engine kvůli jednoduchosti, výkonu a dostupnosti systému DOTS, který dovoluje využít potenciál všech jáder procesoru na 100% při vykreslování scény a tak dosáhnout lepších výsledků.

Existují i jiné možnost pro vizualizaci grafu, ale většina těchto nástrojů jsou jen dvoudimenzionální. Nejoblíbenější volně dostupný nástroj se jmenuje Gephi [6]. Jako další velice učinným nástrojem je Neo4j [7], který lze taktéž použít jako JavaScriptová knihovna kombinovaná s ReactJs [8].

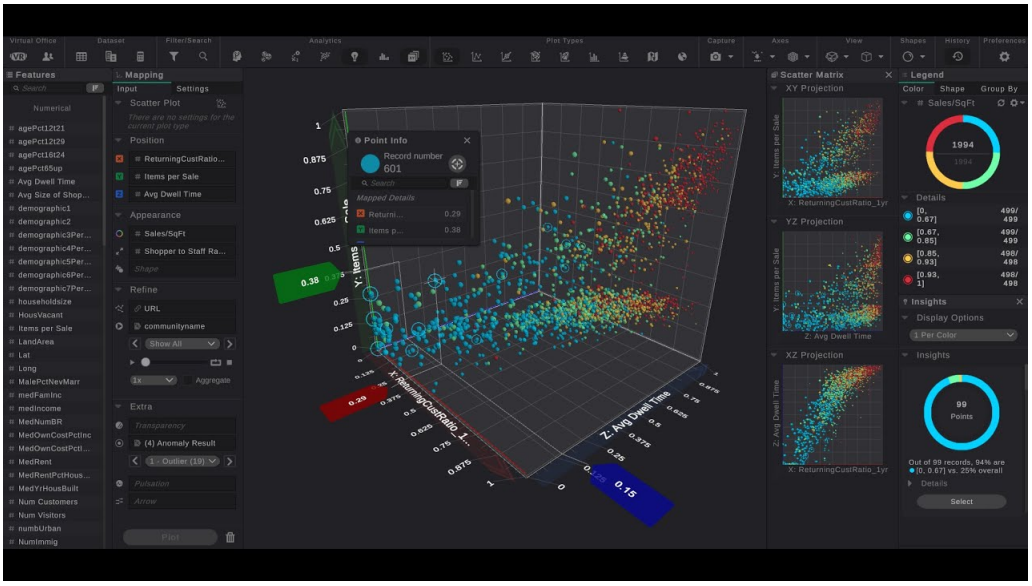
	Realtime vizualizace	Platforma	Licence	Výpočet vrstvy	Rozšířitelnost	API	VR
Gephi	✓		open-source	✓	✓	✓	✗
Graphviz	✓		open-source	✓	✗	✓	✗
Tableau	✓		shareware	✓	✓	✓	✗
Cytoscape	✓		open-source	✓	✓	✓	✗
Neo4j Bloom	✓		open-source	✓	✓	✓	✗
Polinode	✓		freemium	✓	✓	✓	✗
Virtualitics	✓		annual	✓	✓	✓	✓

Tabulka 2.1: Vlastnosti ostatních aplikací

2.1 Virtualitics

Cílem Virtualitics [5] je utvářet lepší svět umožněním chytřejšího rozhodování, vizualizací a spolupráci na základě dat prostřednictvím umělé inteligence. Tato pokročilá technologie je založená na desetiletí výzkumu ohledně umělé inteligence a vizualizaci dat.

Virtualitics se skládá ze tří části, kde každá část slouží k jiné specifické funkcionalitě. **Virtualitics Immersive Platform (VIP)** je určená k vizualizaci multidimenzionálních grafů dle poskytnutých dat bez nutnosti programovacích schopností. Vizualizovat data lze taktéž na mapě světa, ale i ve virtuální realitě s možností kooperace s ostatními uživateli.



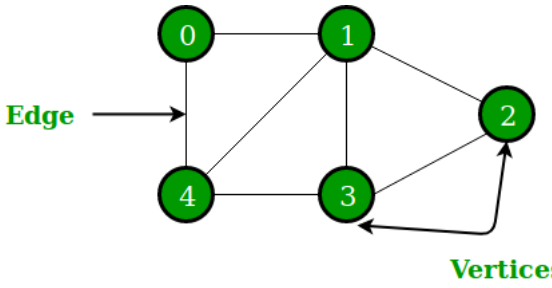
Obrázek 2.1: Virtualitics

Kapitola 3

Teorie grafů

Mnoho věcí na světě by neexistovalo, pokud by zde nebyl problém, který si vyžaduje řešení. Tento fakt platí na vše, ale převážně v oboru informační technologie. V případě, že byla potřeba udržet data pohromadě, tak se pro patřičný problém hledala ta nejvhodnější datová struktura. Proto vznikly grafy, ale ty grafy, které nemají žádný kořen, spíše mají nebo nemají směr. Ovšem věci jsou komplikovanější, protože i hrany grafu mohou i nemusí mít směr. Grafy existují proto, protože si lidé potřebovali vyobrazit vzorec nebo trend nějakých bodů v prostoru. [9]

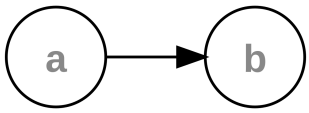
Platí zde jedno pravidlo, a to je, že graf musí mít alespoň jeden uzel, stejně jako platí, že strom musí mít kořen. Podobně tak platí, že graf musí mít alespoň jeden uzel aby mohl být zván grafem. Graf, který má jen jeden uzel je zván **Singleton Graph**. Jak je však známo, uzly, jinak zvány i vrcholy jsou propojené pomocí hran, které mohou být taky nazývány jako linky, nebo vztahy. [9]



Obrázek 3.1: Jednoduchý neorientovaný graf [10]

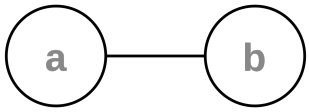
Různé typy hran jsou velmi důležité, pokud jde o rozpoznávání a definování grafů. Ve skutečnosti je to jeden z největších a nejzřetelnějších rozdílů mezi grafy: typy hran, které má. Ve většině případů mohou mít grafy dva typy hran: hranu, která má směr nebo tok, a hranu, která nemá žádný směr ani tok. Označujeme je jako orientované a neorientované hrany. Orientovaná hrana spojuje dva uzly velmi specifickým způsobem. V níže uvedeném příkladu je uzel A spojen s uzlem B orientovanou hranou; existuje jen jeden způsob cestování mezi těmito dvěma uzly - pouze jedním směrem, kterým můžeme jít. Je docela běžné označovat uzel, ze kterého začínáme, jako počátek a uzel, do kterého

cestujeme, jako cíl. V orientované hraně můžeme cestovat pouze z místa počátku do cíle, nikdy ne naopak.



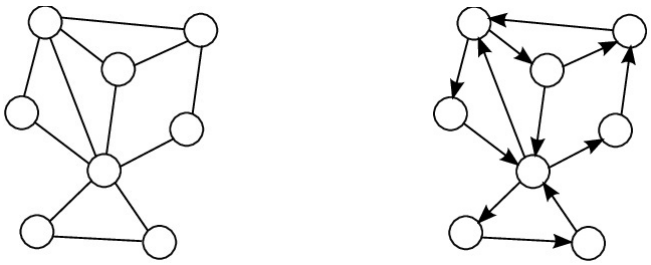
Obrázek 3.2: Orientovaná hrana

U neorientované hrany je to jinak, protože můžeme cestovat oběma směry. To znamená, že cesta mezi dvěma uzly je obousměrná, což taktéž znamená, že počáteční a cílové uzly nejsou pevné.



Obrázek 3.3: Neorientovaná hrana

Tato diferenciace je ve skutečnosti docela důležitá, protože hrany v grafu určují, jak se graf nazývá. Pokud jsou všechny hrany v grafu orientované, říká se, že graf je orientovaný, také nazývaný digraf (od spojení directed graph). Pokud jsou všechny hrany v grafu neorientované, grafu se říká neříká jinak, než neorientovaný graf nebo jednoduše graf.



(a) Neorientovaný graf (b) Orientovaný graf

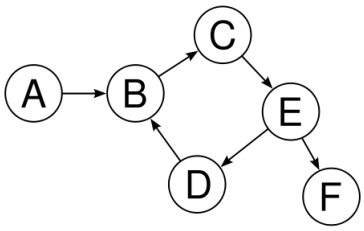
Obrázek 3.4: Typy grafů

Definice grafu je taková:

$$G = (V, E) \tag{3.1}$$

Kde V (vertices) je množina uzlů a E (edges) je množina hran.

Pro neorientovaný graf platí, že množina hran nemusí mít správně řazené počátky a cíle, vzhledem k tomu, že cestovat můžeme oběma směry, ale u orientovaného grafu je řazení důležité. Příkladem může být graf na obrázku 3.5:

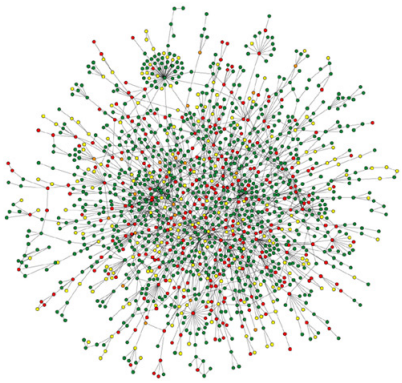


Obrázek 3.5: Příklad orientovaného grafu [11]

Je možné cestovat z uzlu A do B, ale nemůžeme cestovat z uzlu B do A, kdyby směr hrany byl zaměněný, tudíž řazení počátku a cíle bylo obráceně, nebylo by možná cestovat kamkoliv z bodu A.

3.1 Komplexní sítě

V kontextu teorií sítí je komplexní síť graf s netriviálními topologickými rysy. Rysy, které se nevyskytují v jednoduchých sítích, jako jsou mřížky nebo náhodné grafy, ale často se vyskytují v sítích představujících skutečné systémy. Studium komplexních sítí je do značné míry inspirováno empirickými poznatky skutečných sítí, jako jsou počítačové sítě, biologické sítě, technologické sítě, mozkové sítě, klimatické sítě a sociální sítě. [12] [13] [14]



Obrázek 3.6: Komplexní síť [15]

Síť je relační forma reprezentace diskrétních dat. Dva nejdůležitější koncepty týkající se sítí jsou entity a vztahy mezi nimi. Entity se označují jako uzly a vztahy se označují jako hrany. Když je to nezbytné, reprezentujeme uzly a hrany přidáním vlastností známých také jako atributy. Vztah nebo hrana obvykle zahrnuje dvě diskrétní entity nebo uzly, ačkoli entita může být ve vztahu sama se sebou, takový vztah se označuje jako **reflexivní** [16].

Komplexní síť poznáme díky jeho netriviální struktůře. Síti s triviální struktůrou říkáme **Klasická síť**. Příkladem klasické sítě je lineární síť - časová osa našeho života, každá životní událost (jako „narození“, „první chůze“, „promoce školy“, „manželství“, atd.) je entitou s alespoň jedním

atributem, kterým je čas. „Děje se po“ je vztah v tomto scénáři, protože hrana spojuje dvě události společně s jednou událostí, která se odehrává bezprostředně po druhé. Důvod, proč je tato síť považována za jednoduchou, je skutečnost, že má pravidelnou strukturu. Složitá síť má netriviální strukturu, není to ani mřížka, ani strom. Tyto složité sítě se vyskytují v přírodě a v umělém světě v důsledku decentralizovaných procesů bez globální kontroly. Mezi zástupce této třídy sítí patří [16]:

- **Sociální sítě** - Přátelé, sledující, atd.
- **Kulturní sítě** - Náboženské sítě, jazykové rodiny atd.
- **Technologické sítě** - Dopravní a komunikační systémy atd.
- **Finanční sítě** - Mezinárodní obchod atd.
- **Biologické sítě** - Interakce genů/proteinů, epidemie nemocí atd.

Kapitola 4

Herní Engine

Pokud jde o vývoj her, je dnes výhodnější použít již existující herní engine než vytvář vlastní, vzhledem k časové náročnosti. Herní enginy jsou původně určeny k vývoji videoher. Dnes se ovšem využívají i v jiných odvětví, jako je například vizualizace produktů, simulacím v reálném čase, nebo i ve filmovém průmyslu, ovšem i při vývoji nástrojů pro virtuální realitu. Mezi nejznámější herní enginy patří Unity [1], Unreal Engine [17], Game Maker [18] a Godot [19]. V rámci této práce se budeme zabývat vizualizací grafů v herním enginu Unity.

4.1 Historie Unity

Herní engine Unity byl spuštěn v roce 2005 a jeho cílem bylo „demokratizovat“ vývoj her tím, že jej zpřístupní více vývojářům [20]. Příští rok byla společnost Unity označena jako druhá v kategorii Nejlepší využití grafiky Mac OS X v Apple Design Awards společnosti Apple Inc. [21] Unity byl původně vydán pro Mac OS X, později přidal podporu pro Microsoft Windows a webové prohlížeče. [22]

Unity 2.0 byl vydán v roce 2007 s přibližně 50 novými funkcemi. Vydání zahrnovalo optimalizovaný terénní engine pro detailní 3D prostředí, dynamické stíny v reálném čase, směrová světla a další funkce. Vydání také přidalo funkce, díky kterým mohli vývojáři snadněji spolupracovat. To zahrnovalo síťovou vrstvu pro vývojáře, kteří mohli vytvářet hry pro více hráčů. [23]

Unity 3.0 který se objevil v září 2010 přinesl funkce rozšiřující grafické funkce enginu pro stolní počítače a herní konzole, mimo to přišla i jeho podpora pro mobilní operační systém Android. [24]

The Verge k vydání Unity 5 v roce 2015 uvedl: „Unity začalo s cílem univerzálně zpřístupnit vývoj her. Unity 5 je dlouho očekávaným krokem k této budoucnosti.“ [25] Unity 5 vylepšil osvětlení a zvuk. Prostřednictvím WebGL mohli Unity vývojáři přidávat své hry do kompatibilních webových prohlížečů, aniž by hráči museli používat zásuvné moduly. Unity 5 nabízí globální osvětlení v reálném čase, náhledy světelných map, Unity Cloud, nový zvukový systém a fyzikální engine Nvidia PhysX 3.3. [26]

V prosinci 2016 společnost Unity Technologies oznámila, že změní systém číslování verzí pro Unity z identifikátorů založených na sekvenci na rok vydání, aby sladily verzi s jejich častější kadencí vydání; Po verzi 5.6 následovala verze 2017. [27]

4.2 Universal Render Pipeline (URP)

Cílem Universal Rendering Pipeline (URP) je poskytnout optimalizovaný výkon pro vývojáře zaměřené na širokou škálu platforem, VR a her s omezenými potřebami osvětlení v reálném čase. Funguje tak, že dělá nějaké kompromisy, pokud jde o osvětlení a stínování. URP provádí jednoprůchodové dopředné vykreslování s jedním stínovým světlem v reálném čase. Ve srovnání s klasickým renderovacím enginem Unity (Legacy Pipeline) URP neprovádí další draw call každého světla v okolí. URP je také výhodný v tom, že podporuje nástroj zvaný Shader Graph (4.2.2). [28]

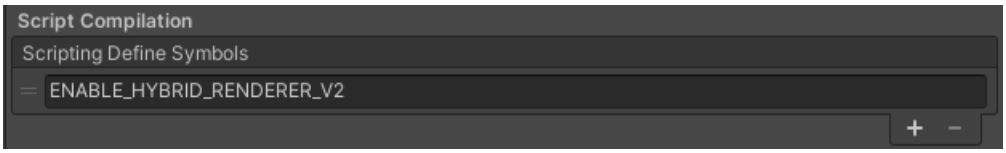
URP je součástí Scriptable Render Pipeline (SRP), avšak zde existuji i možnosti jako je High Definition Render Pipeline (HDRP) [29] nebo Lightweight Render Pipeline (LWRP) [30], který je dostupný ve starších verzích Unity, dnes však nahrazený URP, ale ani jedna z těchto možností není vhodná pro aktuální projekt především kvůli nedostatečné functionalitě, nebo vyšší hardwarové náročností.

Díky tomu, že je URP postavený tak, aby poskytl optimalizovaný výkon, není třeba silnějšího hardwaru oproti HDRP. Hardwarové požadavky lze zjistit na oficiálních stránkách Unity [31].

4.2.1 Hybrid Renderer V2

Hybrid Renderer poskytuje systémy a komponenty pro vykreslování entit ECS (7.4). Nejedná se však o Render Pipeline, ale o systém, který shromažďuje data nezbytná pro vykreslování entit ECS, a odesílá je do stávající architektury vykreslování Unity. Universal Render Pipeline (URP) a High Definition Render Pipeline (HDRP) jsou zodpovědné za vytváření obsahu. [32]

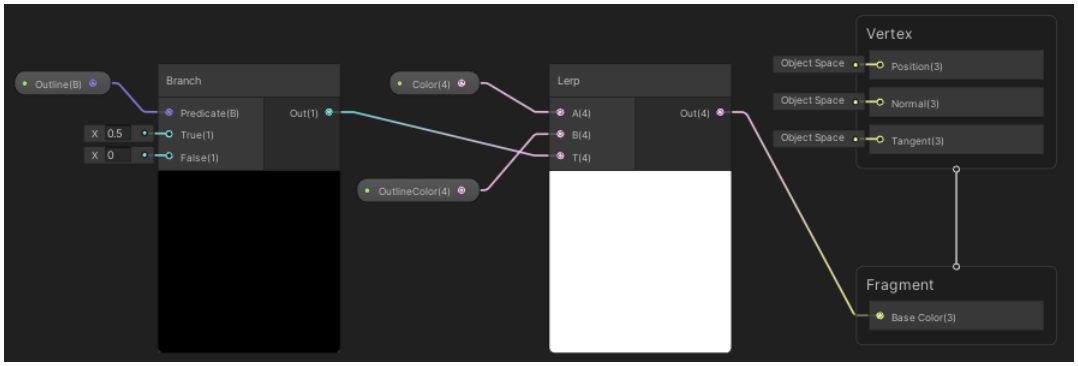
Pro povolení Hybrid Rendereru V2 je třeba přidat symbol do skriptovacího preprocesoru v nastavení projektu.



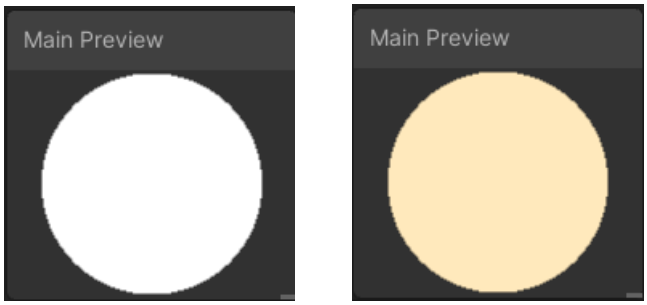
Obrázek 4.1: Symbol pro skriptovací preprocesor pro povolení Hybrid Rendereru V2

4.2.2 Shader Graph

V předchozích verzích Unity bylo jediným způsobem, jak vytvořit materiálové shadery tím, že se musely naprogramovat nebo použít externí zásuvný modul. Unity od vydání verze 2018 přinesl svůj nativní nástroj k těmto potřebám, který lze použít k vytváření vizuálně složitých shaderů pomocí uzlů. Na rozdíl od procesu psaní kódu, ukládání, kompilace a testování v editoru je Shader Graph schopný ukázat, co se s materiálem děje v reálném čase. [33]



Obrázek 4.2: Shader pro uzly grafu



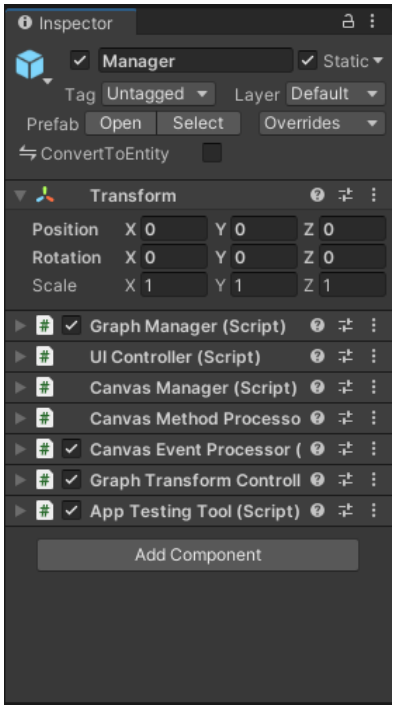
(a) (b)

Obrázek 4.3: Náhled uzlu před a po označení

4.3 Skriptování v Unity

Unity skripty jsou primárně psány v jazyce C#. Kdysi byla možnost využití jazyka JavaScript i Boo, ale Unity se časem začal rozšiřovat takovým způsobem, že nebylo možné nadále se starat o tyto dva jazyky, a proto přestaly být podporovány. Unity funguje na základně toho, že scéna obsahuje objekty zvány **Game Object**, které v sobě drží skripty zvané **Component** [34]. Jakákoliv serializovatelná proměnná lze viděl a měnit v inspektoru. V případě systému DOTS jsou tyto objekty proměněny do entit.

Unity má zarezervované názvy metod, které když jsou definovány ve skriptu, budou automaticky volány ve správný čas, dle tzv. **Execution Order** [35].



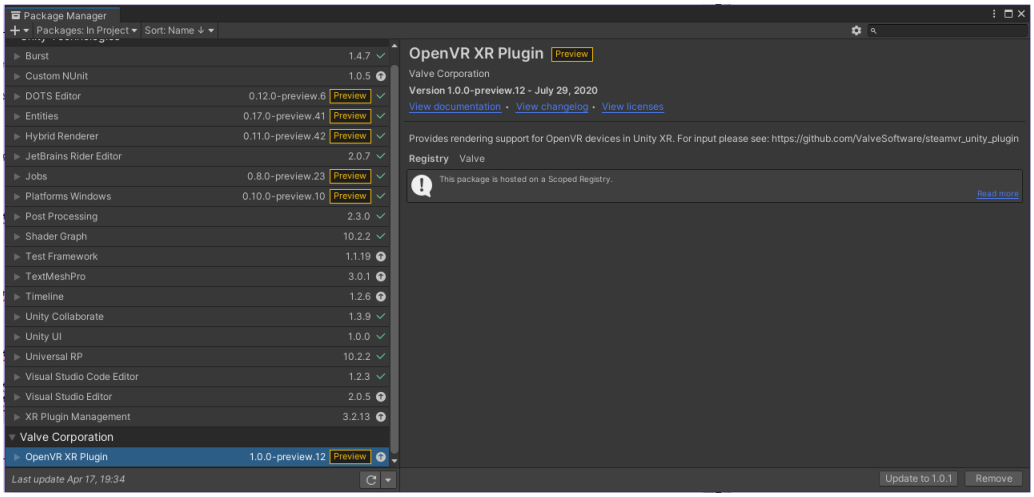
Obrázek 4.4: Unity Inspector

Kapitola 5

Vývoj aplikací pro virtuální realitu

S nástupem virtuální reality má Unity (od verze 2019.3) rovněž podporu pro tento typ zařízení. Podporuje řadu platforem mezi které patří např. SteamVR, Oculus, Playstation VR, Microsoft HoloLens, atd. Tato aplikace je postavena na platformě SteamVR, konkrétně pro zařízení HTC Vive.

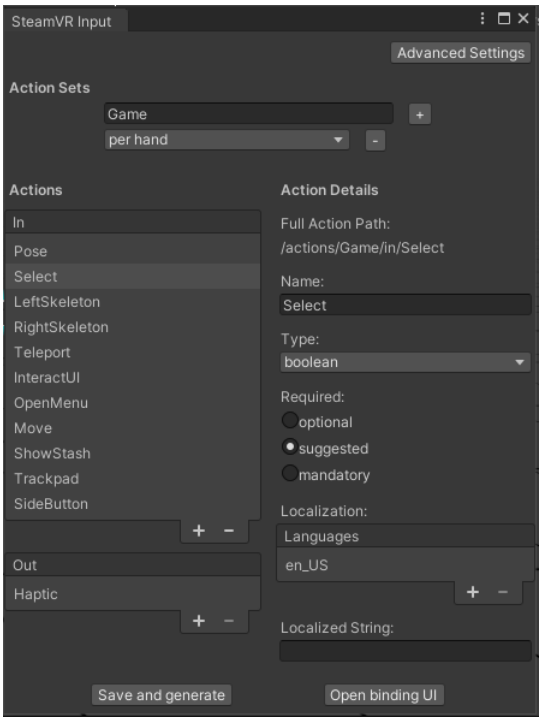
Jako první bylo třeba nainstalovat OpenVR XR plugin pomocí Package Manageru. Po instalaci tohoto pluginu lze použít prefaby[1], které jsou již vytvořené ve SteamVR, tyto prefaby slouží k vytvoření kamery pro virtuální realitu. Všechny tyto prefaby lze nalézt ve složce Prefabs.



Obrázek 5.1: Package Manager

OpenVR plugin má taktéž předchystané skripty pro teleportaci, ale aplikace používá vlastní skripty pro tyto účely. Jako další je třeba definovat akce pomocí SteamVR Input dialogu v Unity, který vygeneruje skript obsahující statické konstantní proměnné těchto akcí s předponou *SteamVR_Action_* a jménem akce.

[1] *Prefab* - Předchystané objekty k opakovanému použití

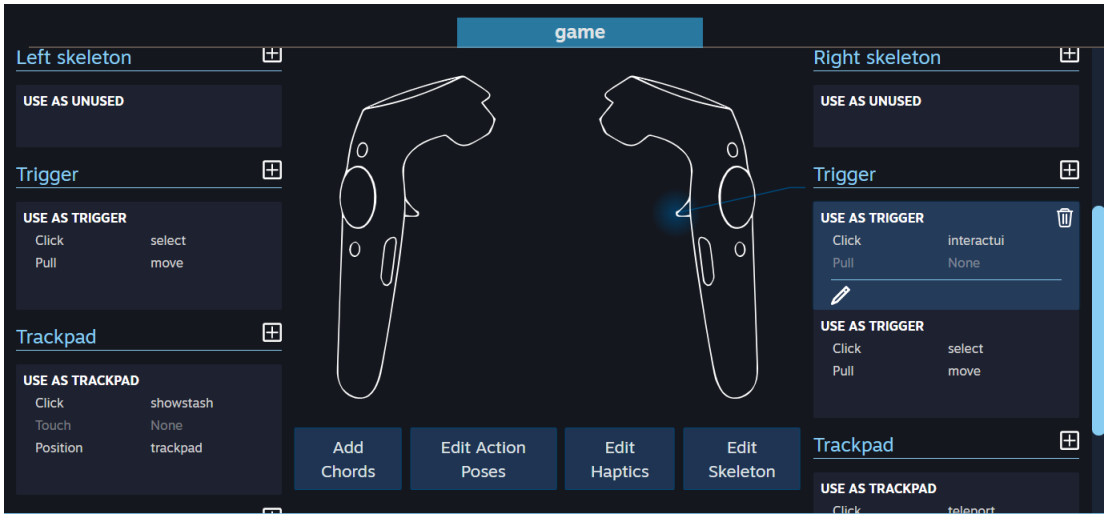


Obrázek 5.2: SteamVR Input Dialog

Po vytvoření akcí je třeba tyto akce nastavit na příslušná tlačítka ovladače. Ovladače jsou různé pro každý headset, avšak aplikace má přednastavené ovládání pro HTC Vive a Oculus Rift (přesněji Oculus Quest 2, ale pro SteamVR známo jako Oculus Rift), protože obě tyto zařízení byla použita pro testování aplikace.

5.1 Nastavení ovládaní ve SteamVR

Pro správnou funkčnost aplikace bylo třeba nastavit příslušným tlačítkům jejich funkce. K tomu slouží **Bindings UI** ve SteamVR. Tato aplikace taktéž podporuje přepínání mezi ovladači, tudíž lze nastavit ovladače i pro zařízení, které vývojář nevlastní.



Obrázek 5.3: Nastavení ovladačů ve SteamVR

Kapitola 6

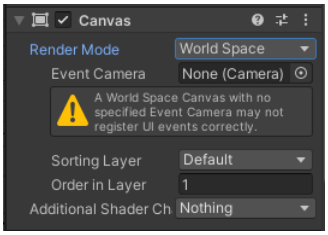
Vývoj aplikace k vizualizaci grafu

Projekt byl prvně založen ve verzi **2019.2.18f1**, kdy byla implementace SteamVR řešena prostřednictvím doplňku z Unity Asset Store, později byl projekt převeden do vyšší verze Unity **2020.2.1f1** kvůli oficiální podpoře virtuální reality jak již bylo zmíněno ve vývoji aplikaci pro virtuální realitu (5).

Implementace systému DOTS (7.4) způsobila, že projekt musel být převeden do URP (4.2) kvůli podpoře Hybrid Rendereru V2 (4.2.1), který se využívá k vykreslování uzlů grafu.

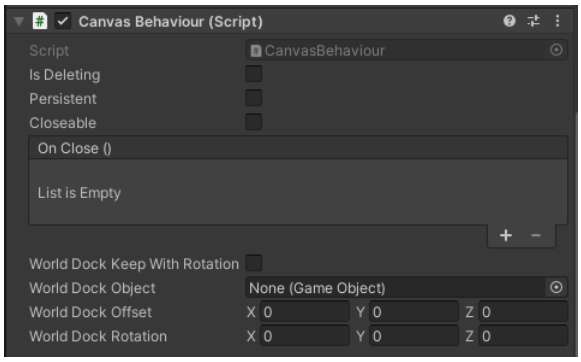
6.1 Implementace uživatelského rozhraní

Pro uživatelské rozhraní byl použit původní systém uživatelského rozhraní Unity, dále jen UI. Pro každé okno aplikace byl vytvořen *GameObject* s komponentou *Canvas*, který má nastavitelný **Render Mode**, ten nastavuje způsob zobrazení určitého okna, a vzhledem k tomu, že se jedná o tří dimenzionální scénu ve virtuální realitě, je toto nastavení vždy na hodnotě **World Space**.



Obrázek 6.1: Canvas komponenta

Event Camera je automaticky nastavená skriptem po vytvoření okna na hlavní UI kameru, která je vytvořená na pravém ovladači, protože pravý ovladač je určen k interakci s uživatelským rozhraním. Každé okno uživatelského rozhraní také obsahuje komponentu *CanvasBehaviour*, která obsahuje vlastnosti tohoto okna.

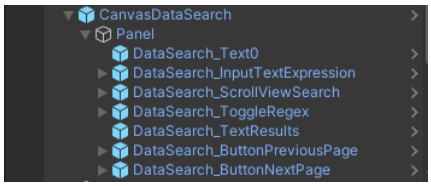


Obrázek 6.2: CanvasBehaviour komponenta

Persistent atribut určuje, zda-li bude okno stále otevřené ikdyž uživatel otevře nové okno. Pokud je tato hodnota nastavená na *false*, okno lze zavřít jen pokud je na něj ukazáno a stlačeno tlačítko pro otevření menu. Toto pravidlo platí tehdy, pokud je atribut *Closeable* nastavené na *true*, jinak se o zavření tohoto okna stará sám systém.

On Close je seznam obsahující metody, které jsou zavolané tehdy, pokud probíhá zavíraní tohoto okna. **World Dock** je atribut určující dokování aktuálního okna na jiné okno, toto dokování je vhodné pokud je třeba udržet jedno okno vedle druhého tak, aby bylo jasné, že dokované okno je závislé na okně, na které je dokované. **Keep With Rotation** zaručuje, že dokované okno bude udržovat stejnou rotaci jako okno, na které je dokované a tak bude stále vedle něj. *Offset* a *Rotation* jsou další nastavitelné atributy, které určují rozestup pozice a rotace od okna, na které je vázané.

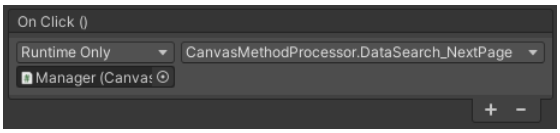
O otevírání a zavírání oken se stará statická třída *UISystem*, která zároveň obsahuje statické metody i pro vyhledávání obsahu aktuálního nebo jiného okna. Všechna okna musí mít v názvu předponu *Canvas* a všechny tlačítka, textové pole, atd. musí mít ve jméně jako předponu název okna bez předpony *Canvas*, např. okno s názvem *CanvasDataSearch* má textové pole s názvem *DataSearch_InputTextExpression*. Je zároveň třeba udržovat v názvu typ objektu kvůli vyhledávání obsahu pomocí statické třídy *UISystem*.



Obrázek 6.3: Okno DataSearch v hierarchii scény

Pokud je jakékoliv okno otevřeno nebo uzavřeno, vyhledává se metoda *<Název okna>_Open* nebo *<Název okna>_Close* v komponentě *CanvasEventProcessor* objektu *Manager* a pokud je nalezena, je automaticky zavoláná. Díky tomu lze provést nějakou akci než uživatel bude moci s tímto oknem pracovat. Zároveň po otevření okna je vyhledávána metoda *<Název okna>_Update* a v případě existence této metody je volán každý snímek po dobu, kdy je toto okno otevřeno.

Většina metod akcí tlačítek je nadefinovana v komponentě *CanvasMethodProcessor* objektu *Manager*, a tyto metody jsou volány tehdy, pokud je tlačítko stisknuto, nebo se udála jiná událost jiného objektu. V názvu událostní metody se udržuje název okna jako předpona, např. *DataSearch_NextPage*.



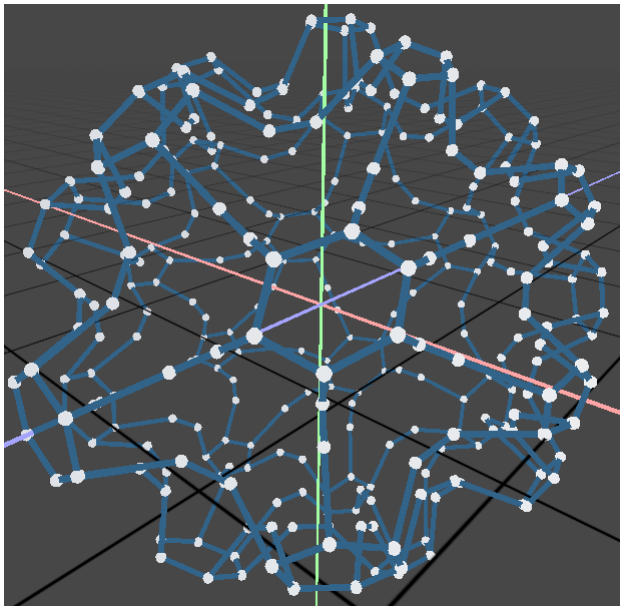
Obrázek 6.4: Seznam metod události po stisknutí tlačítka

6.2 Implementace grafu

Všechny uzly a hrany patří pod jeden graf, a tento graf je v aplikaci objektem třídy *GraphModel*. Komponenta *GraphManager* objektu *Manager* udržuje seznam všech grafů aplikace a také aktuální graf. Každý graf může mít odlišné nastavení oproti ostatním, proto se toto nastavení ukládá v objektu grafu, nikoliv v manažeru grafů. Toto nastavení je reprezentováno třídou *GraphModelProperties*, a obsahuje jen ty nejdůležitější vlastnosti grafu, jako je *Zoom*, *Translation*, *Rotation*, *Scale* a *Offset*. Každá z těchto atributů mění výslednou vizualizaci grafu, vzhledem k tomu, že uzly a hrany, které patří pod tento graf jsou podřízené jeho vlastnostem. *Zoom* grafu bývá většinou automaticky vypočítán tak, aby byl vhodný k práci.

Jak již bylo zmíněno, uzly a hrany patří pod jeden graf, a proto každý objekt grafu udržuje seznam všech těchto uzlů a hran a kdykoliv dojde k aktualizaci jakékoliv vlastností téhož grafu, tak jsou aktualizovány vlastnosti těchto udržovaných objektů.

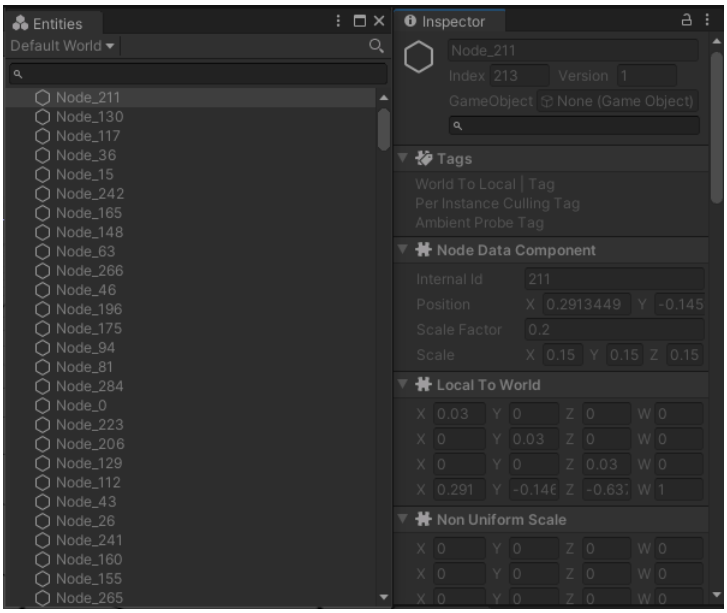
Původní implementace grafů ukladala jen informace o zoomu a offsetu grafu, vzhledem k tomu, že nebyl použit systém DOTS (7.4) a proto informace o pozici, rotaci a velikosti byly zapisovány v *Transform* komponentu grafu, tudíž všechny objekty pod tímto grafem byly automaticky transformovány. Ale po implementaci systému DOTS nejsou uzly a hrany reprezentovány jako *GameObject* v hierarchii scény ale jako *Entity*, které nejsou zobrazeny v hierarchii scény a tedy se jedná o objekty, které nemohou být potomky objektu grafu, tudíž se nemohou řídit podle transformačních pravidel toho objektu. Proto tyto informace byly přesunuty do třídy *GraphModelProperties*.



Obrázek 6.5: Jednoduchý graf

Předtím než byl implementován systém DOTS byl zde pokus o implementaci jedné části DOTS,

a to je Job system (7.4.1), který se měl starat o aktualizaci uzlů a hran ve více vláknech než v hlavním, protože aplikace tehdy obsahovala animace při pohybu těchto uzlu a hran v případě, že se měnil zoom grafu nebo se provedl výpočet rozložení grafu a to zapříčinilo změnu pozic všech uzlů. Později však tato animace byla odstraněna a všechny uzly grafu byly převedeny do systému DOTS. Díky této změně byla aplikace schopna zvládnout mnohonásobně více uzlů na scéně než předtím.



Obrázek 6.6: Hierarchie ECS entit

	GameObject Architektura	Entity Component System
100	200 FPS	288 FPS
200	166 FPS	238 FPS
1000	59 FPS	163 FPS
4000	15 FPS	112 FPS

Tabulka 6.1: Měření snímkové frekvence při různém počtu uzlů grafu

Zoom hodnota grafu se vypočítává automaticky hned po importu grafu nebo při použití funkce v hlavním menu. Výpočet se provádí pomocí aktuální výšky uživatele a největší hodnoty z velikosti grafu.

$$Zoom = \frac{Height}{Max} \tag{6.1}$$

6.3 Implementace uzlu

Uzel je nejdůležitější části grafu, bez uzlu by hrany neměly smysl, a bez hran nemůžeme vytvořit žádnou síť. Uzel je objektem třídy *GraphNode*, který může i nemusí být udržován v objektu grafu. Pokud uzel nepatří žádnému grafu, jedná se o uzel *nepřipravený*, a tudíž se nezobrazuje. Takový uzel může vzniknout tehdy, pokud uživatel spustil import ve formátu, který uzly zakládá, ale nepřiřazuje graf. Nakonec vždy uzel bude aspoň jednou přiřazen grafu, a tím přejde do stavu *připraveného*.

Třída *GraphNode* dědí z třídy *GraphEntity*, která definuje dvě identifikační proměnné jako je *InternalId* a *Id*. *InternalId* je identifikační číslo objektu v aplikaci, které je součástí hash kódu objektu a předává se při komunikaci mezi aplikaci a API (6.12), tudíž se jedná o konstatní hodnotu, která je automaticky přidělena objektu po vytvoření. *Id* je proměnná, která je buď automaticky vygenerována, nebo přidělená při importu grafu, pokud daný formát grafu toto identifikační číslo podporuje. Toto identifikační číslo je měnné, vzhledem k tomu, že se může později změnit, pokud uživatel chce toto identifikační číslo tvořit jinak pomocí nástroje **Graph Builder**. Každý objekt třídy *GraphEntity* má slovník *Data*, kde klíčem je textový řetězec a hodnotou jakýkoliv objekt jakéhokoliv typu, to znamená, že každý uzel a hrana může mít uložena data pod nějakým klíčem.

Uzel je objekt grafu, který může mít libovolnou pozici v grafu a proto existuje proměnná *RealPosition*, která udržuje třídimenzionální hodnotu reálné pozice v grafu. Reálná pozice je ta pozice, která byla načtena ze souboru, ale ve výsledku uzel bude vykreslen na jiné pozici, protože graf je před použitím přibližen nebo oddálen tak, aby velikost grafu byla vhodná pro práci.

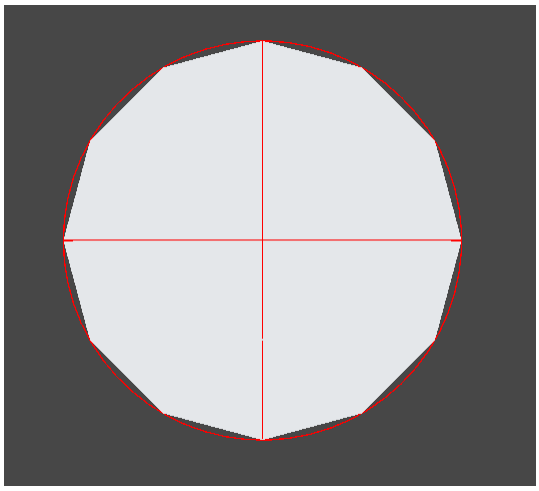
Aktuální pozici uzlu lze zjistit pomoci getteru *TransformPosition*, který provádí výpočet za pomocí reálné pozice uzlu (*RealPosition*), reálného posunu grafu (*Translation*), offsetu grafu (*Offset*), zoomu grafu (*Zoom*), rotaci grafu (*Rotation*), a velikosti grafu (*Scale*). Poté co je pozice vypočtena se provede akce, která zamění hodnoty Y a Z, pokud uživatel v konfiguraci nastavil, zda-li výšková osa je Z nebo Y. Původní výšková osa v Unity je Y, to samé platí i v této aplikaci.

$$\begin{aligned} Position &= (RealPosition + RealTranslation + Offset) \times Zoom \\ RotatedPosition &= Rotation \times (Position - Offset \times Zoom) + Offset \times Zoom \\ TransformPosition &= RotatedPosition \times Scale \end{aligned} \tag{6.2}$$

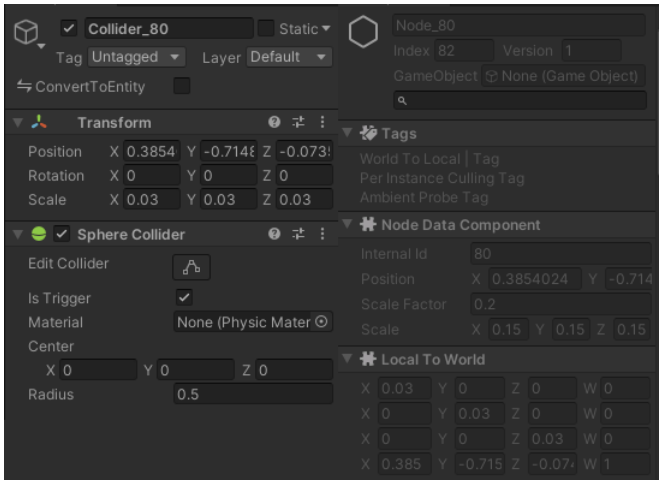
Velikost uzlu je vypočítaná pomocí proměnné *Size* a hodnoty konfiguračního atributu *NodeSizeMultiplier*. Tento atribut lze změnit v hlavním menu aplikace. Pokud import grafu nenastavuje velikost uzlů, nebo není tak nastaveno pomocí Graph Builderu, velikost uzlů je původně nastavená na hodnotu konfiguračního atributu *DefaultNodeSize* což je *0.15* metrů.

Při převodu uzlu do ECS (7.4) vznikl problém, kvůli kterému nešlo označovat uzly ovladačem, protože systém DOTS využívá vlastní fyzikální engine, a proto raycast, který je vysílán z ovladače není schopný detekovat kolizní komponenty jiného systému. Řešení je takové, že se při vytváření

ECS uzlu zároveň vytváří i uzel typu GameObject, který obsahuje jen kolizní komponentu, který je automaticky vždy přesunut na aktuální pozici uzlu *TransformPosition*. To znamená, že na jednom místě jsou doopravdy dva objekty, kde jeden, který je registrován jako entita ECS a určen jen k renderování, a druhý je registrován jako GameObject a určen jen jako kolizní maska. Ve výsledku toto řešení nemělo žádný vliv na výkon aplikace a problém byl tak vyřešen.



Obrázek 6.7: Kolizní maska uzlu



Obrázek 6.8: Inspektor uzlu

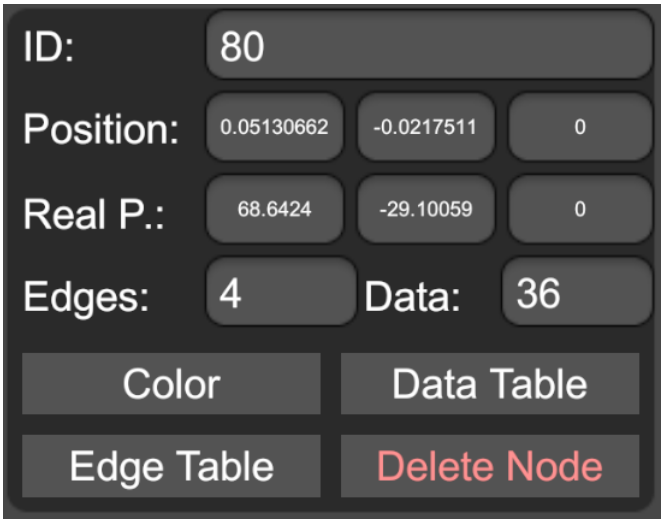
Is Trigger slouží k tomu, zda-li se jedná o kolizi pevnou, to znamená, že za normálních okolností kdybychom vytvořili objekt, s pevnou kolizní maskou, který padá dolů přímo na tento objekt taktéž s pevnou kolizí, neměl by padající objekt propadnout, ale v případě, že se jedná o nepevnou kolizi, tudíž **Is Trigger** je zaškrtnutý, padající objekt propadne. Důvod tohoto atributu je jen pokud chceme zjistit, zda-li je nějaký padající objekt v prostoru tohoto objektu, v našem případě, zda-li

raycast prochází tímto prostorem, a pokud ano, označíme uzel, kterému patří tento prostor. Pokud by **Is Trigger** nebyl zaškrtnutý, uživatel by mohl použít teleportační tlačítko na svém ovladači a postavit se na tento uzel.

Uzel taktéž může mít vlastní barvu, je však pravidlem, že uzel může být jednobarevný, ale jeho proměnná *Color* je pole, které může mít až dvě hodnoty. Je to kvůli tomu, že uzel může mít barvu určenou dle vzdálenosti kamery od uzlu. Pokud je proměnná *Color* nastavena na pole s jednou hodnotou, tato barva bude vždy konstantně aplikována na uzel, ale pokud toto pole bude mít dvě hodnoty, provádí se výpočet barvy dle vzdálenosti kamery od uzlu, čím blíž kamera je, tím více se barva bude blížit první barvě v poli, a naopak. Je taktéž určený interval, ve kterém se tento výpočet provádí. Původní interval je <1, 20> metrů s ohledem na zoom grafu, tudíž při polovičním zoomu je interval <0.5, 10>. Tento interval je možno změnit v konfiguraci pod klíčem *ColorDistance*, kde *ColorDistance* je dvoudimenzionální vektor, ve kterém X je minimum a Y je maximum.

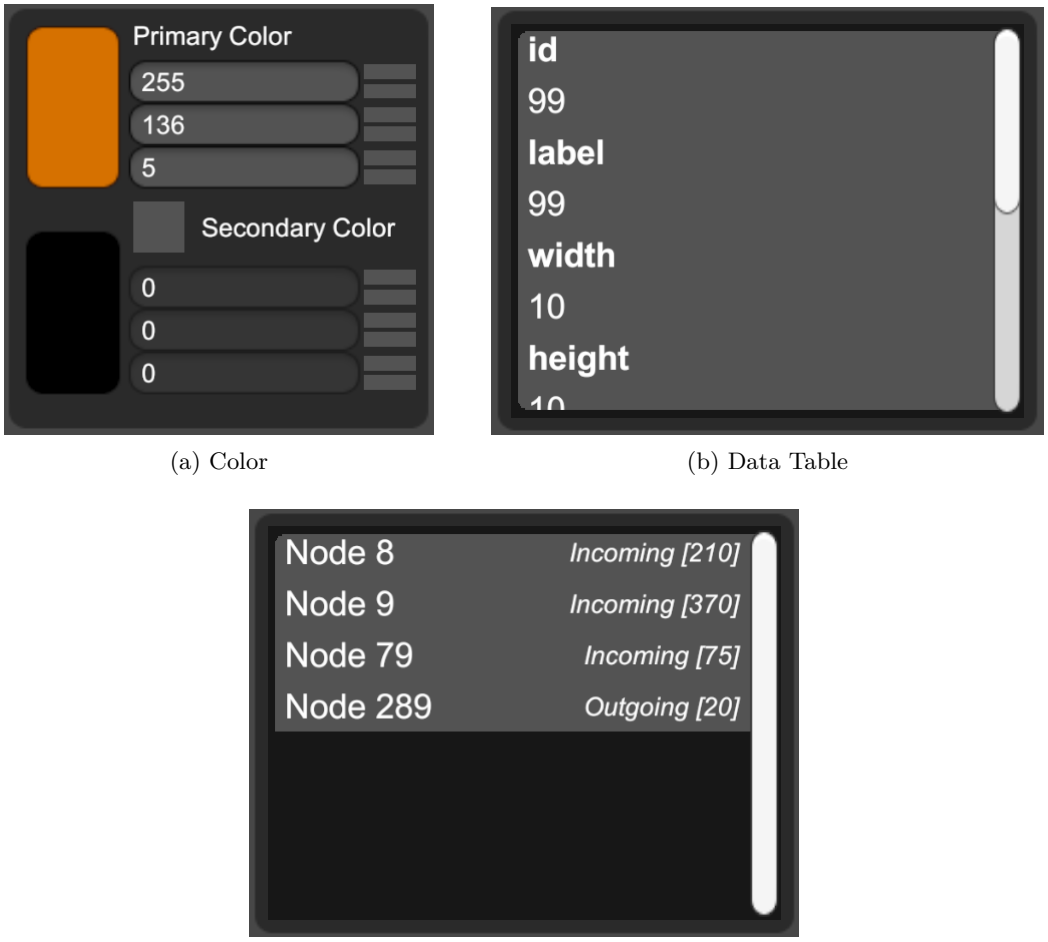
$$ResultColor = \begin{cases} Color_0, & \text{if } Distance \leq ColorDistance_x \\ Color_1, & \text{if } Distance \geq ColorDistance_y \\ Color_0 - \frac{Color_0 - Color_1}{ColorDistance_y - ColorDistance_x} \times Distance, & \text{else} \end{cases} \tag{6.3}$$

Pokud je uzel v aplikaci označen, bude otevřeno okno s detaily uzlu, ve kterém lze vidět Id, aktuální pozici, realnou pozici, počet hran a počet klíčů v datovém slovníku. V případě označení více než jednoho uzlu, čehož lze docílit pomocí přidržení spouštěcího (Trigger) tlačítka levého ovladače a označování různých uzlů pomocí pravého spouštěcího (Trigger) tlačítka se zobrazí okno se seznamem vybraných uzlů.



Obrázek 6.9: Okno s detaily vybraného uzlu

Toto okno taktéž disponuje čtyřmi tlačítky **Color**, **Data Table**, **Edge Table** a **Delete Node**. Každé tlačítko plní jistou funkcionalitu. Tlačítko Color otevírá nová okno vedle okna detailu uzlu, ve kterém je až šest numerických polí, které slouží k určení primární a sekundarní RGB barvy uzlu. Tyto dvě barvy vkládají hodnotu do pole *Color* uzlu. V případě odkliknutého zaškrtávacího pole má uzel jedinou barvu a proto sekundární barva bude nedostupná k úpravě. Tlačítko Data Table otevírá okno, které obsahuje seznam všech datových klíčů a hodnot uzlu. Tlačítko Edge Table otevírá okno, které zobrazuje seznam všech sousedních uzlů, kde existuje spojení hranou. Záznam tohoto seznamu obsahuje *Id* uzlu, a typ hrany. Pokud je hrana orientovaná, záznam bude obsahovat text *Incoming* nebo *Outgoing*, podle toho, jestli je vybraný uzel počáteční nebo koncový. Za textem typu hrany se nachází váha hrany, která je vypsána jen tehdy, pokud všechny sousední uzly nejsou spojeny hranou o váze *1*, pokud všechny hrany mají váhu *1*, váha není vypsána. A jako poslední tlačítko Delete Node otevírá dialog, který se dotazuje uživatele, zda-li opravdu chce smazat vybraný uzel.



(a) Color

(b) Data Table

(c) Edge Table

Obrázek 6.10: Další dostupná okna pro uzel

6.4 Implementace hrany

Hrany tvoří síť mezi uzly grafu. Hrana je objektem třídy *GraphEdge*, pro kterou taktéž platí, že dědí ze třídy *GraphEntity*. *InternalId* není rozlišováno mezi typy objektů (uzel nebo hrana), ale sdíleno, to znamená, že pokud bude vytvořen uzel, bude mu přiřazeno interní id 0, poté hrana, které bude přiřazeno interní id 1, a zase uzel, kterému bude přiřazeno interní id 2, a tento proces se opakuje. *InternalId* je *Unsigned 32bitový Integer*, tudíž je možné, že dostupné id dojdou, ale je to velice nepravděpodobné vzhledem k tomu, že je třeba vytvořit přes 4 294 967 295 objektů.

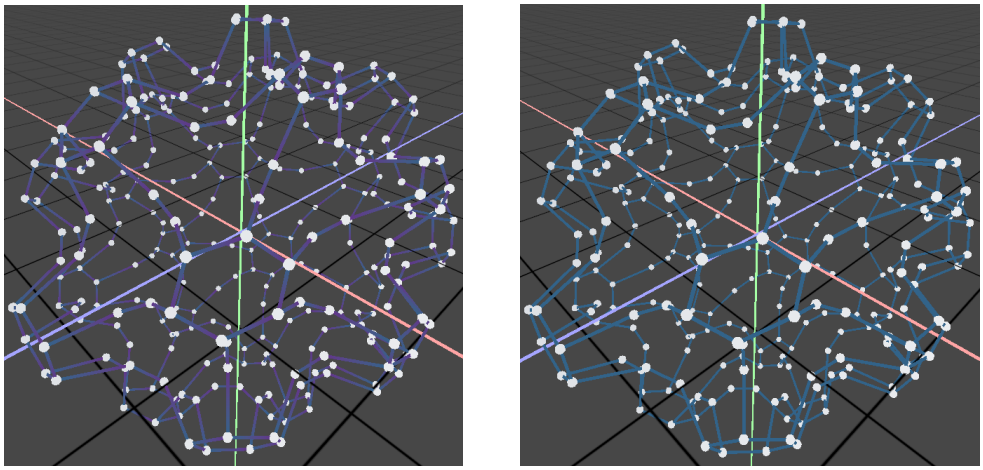
Hrana je od základu tvořena tím, že má zdroj (*Source*) a cíl (*Destination*), většinou díky těmto údajům definujeme, jakým směrem tato hrana míří, ale většina grafů není orientovaná a tím pádem neřešíme, jakým směrem se tato hrana vydává, každopádně je to velice důležitá informace vzhledem k tomu, kdybychom chtěli pomocí těchto hran hledat cestu z jednoho uzlu do druhého. Pokud daná hrana je orientovaná, nelze jim projít ze strany cíle do zdroje. U hran platí stejné pravidlo jako u uzlů, a to je, že mohou být ve stavu *nepřipraveném*. Tento stav vzniká tehdy, pokud je vytvořena hrana, která má zdroj a cíl, ale zdroj, cíl nebo obojí ještě v grafu neexistují. Tento připad může vzniknout jen tehdy, pokud importujeme graf formátem, který dovoluje vytváření hrany dříve než uzly, a do těchto hran se vkládá referenční id uzlů. Po dokončení importu se kontrolují tyto nedokončené hrany, a pokud k těmto hranám jsou nalezeny přislušné uzly, přejdou do stavu *připraveného*, naopak jsou tyto hrany odstraněné, vzhledem k tomu, že není možné k nim najít přislušný uzel/uzly a byly by tak neužitečné. Hrany taktéž obsahují informaci, zda-li jsou orientované (*Directed*), jakou mají váhu (*Weight*) a tloušťku (*Thickness*). Tloušťka hrany je původně vždy nastavená na hodnotu 1, avšak lze ji změnit v **Graph Builderu** dle nějakého atributu ze zdrojového souboru na normalizovanou hodnotu váhy (*Weight*). Výsledná tloušťka hrany je vždy vypočtena pomocí hodnoty tloušťky hrany, základního násobiče tloušťky hrany dle konfigurace pod klíčem *EdgeThickness*, velikosti grafu, a násobiče tloušťky hrany zda-li je označený zdrojový nebo cílový uzel, tento násobič je ve výpočtu využit jen tehdy, pokud je aspoň jeden uzel vybraný a není vybraný zdrojový ani cílový uzel hrany.

$$ConditionedOtherThicknessMultiplier = \begin{cases} OtherThicknessMultiplier, & \text{if } SelectedNodes \geq 1 \\ & \text{and not } SourceSelected \\ & \text{and not } DestinationSelected \\ 1, & \text{else} \end{cases}$$

$$\begin{aligned} TransformThickness = {} & ThicknessMultiplier \times Thickness \\ & \times Scale \\ & \times ConditionedOtherThicknessMultiplier \end{aligned} \tag{6.4}$$

Hrana je vizualizována pomocí *LineRenderer* komponenty v Unity, a vzhledem k tomu, že jeden objekt může mít maximálně jednu tuto komponentu, musí být pro každou hranu vytvořen jeden *GameObject* s komponentou *LineRenderer*. O vytváření těchto objektů a nastavování vlastností těchto čár se stará třída *GraphLine*.

Barva hrany lze měnit v konfiguračním souboru pod klíčem *EdgeStartColor*, *EdgeEndColor*, *EdgeSelectColor*, *EdgeEndConsumerColor* a *EdgeSelectConsumerColor*. *EdgeStartColor* je barva, která začíná od strany zdrojového uzlu, *EdgeEndColor* je barva, která začíná od strany cílového uzlu, *EdgeEndConsumerColor* je barva, která zaačíná od strany cílového uzlu tehdy, pokud se hrana orientovaná, *EdgeSelectColor* je barva po vybrání zdrojového nebo cílového uzlu a *EdgeSelectConsumerColor* je barva, která je aplikovaná jen tehdy, pokud je zdrojový nebo cílový uzel vybraný, hrana je orientovaná a pochází od strany cílového uzlu.



(a) (b)

Obrázek 6.11: Orientovaný a neorientovaný graf

Barva hrany je gradient mezi *EdgeStartColor* a *EdgeEndColor* pokud jde o neorientovaný graf, ale v případě orientovaného grafu je gradient mezi *EdgeStartColor* a *EdgeEndConsumerColor*.

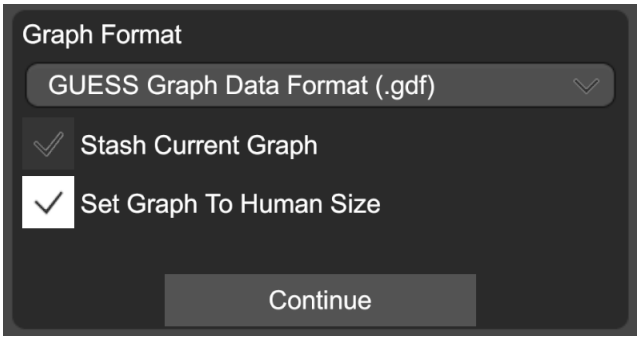
6.5 Implementace importu a exportu grafu

První a nejzákladnější funkce pro začátek s práci je import grafu. Import grafu by měl podporovat více formátů, proto do aplikace byly přidany čtyři základní formáty, jako je DataRelation CSV, tento formát si vyžaduje dva soubory, kde první soubor obsahuje data o uzlech, a druhý soubor, který obsahuje data o hranách. Potom byl implementován formát GDF [36], GEXF [37] a AdjacencyList. V přílohách je minimálně jeden soubor od každého z podporovaných formátů.

Pro jednoduché vytváření formátů existuje abstraktní třída *Abstractformát*, která dědí z abstraktní třídy *AbstractSynchro*. Každý formát dědí ze třídy *Abstractformát*, která obsahuje dvě abstraktní metody *Read* a *Write*, metoda *Read* slouží pro import a *Write* pro export. Vzhledem k tomu, že formát může podporovat více jak jeden soubor, je do parametru posílán identifikační číslo souboru *fileId* (začínající od 0), a jeho *StreamReader* [38] nebo *StreamWriter* [39] podle toho, zda-li uživatel provádí import nebo export grafu.

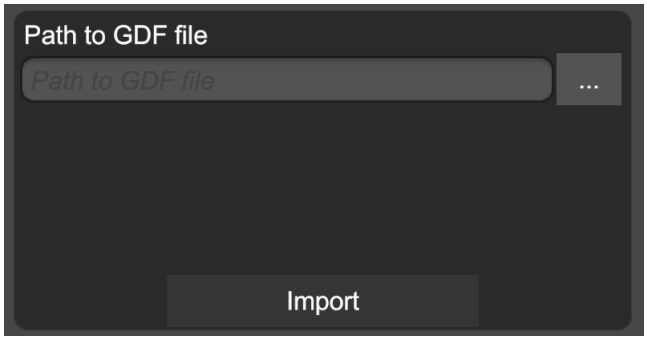
V případě importu grafu existuje metoda *Submit*, která přijímá objekty typu *GraphEntity* nebo *ReaderEntity*. Třída *ReaderEntity* slouží k pozdějšímu vytvoření seznamu názvu atributů ze souboru, který je importován a poté vytvoření objektů grafu. Seznam s názvy atributů je užitečný, pokud uživatel chce využit *Graph Builder* 6.6 a přiřadit jistý zdrojový atribut k objektovému atributu. Metoda *Submit* taktéž kontroluje, zda-li existuje duplicitní objekt, a pokud ano, v případě uzlu nastaví již existujicímu uzlu všechny data z duplicitního uzlu, a v případě hrany, která má stejné *Id* jako *Id* duplicitního objektu, nastaví všechny data již existující hraně z duplicitní hrany (přidá nebo přepíše), ale pokud existuje hrana, která má stejný zdroj a cíl, zvýší váhu již existující hrany o váhu duplicitní hrany.

Pro dokončení již zpracovaného souboru v metodě *Read* je použita metoda *Finish*, která zařídí, že se začne číst další soubor, a pokud již není ve frontě žádný další soubor, všechny reference všech hran jsou vyplňovány a všechny hrany, kterým nebyl nalezen zdrojový nebo cílový uzel jsou smazány. Pokud je atribut *RequiresBuilder* nastaven na *True*, automaticky se otevře **Graph Builder** (6.6) po dokončení importu. Každý formát musí být přidán do slovníku formátů *Graphformáts* v komponentě *GraphManager*, kde klíč bude název formátu a hodnota bude reference typu třídy formátu.



Obrázek 6.12: Import dialog

- **Stash Current Graph** schová aktuálně načtený graf a zobrazí nově načtený graf
- **Set Graph to Human Size** automaticky přiblíží nebo oddálí graf tak, aby výsledný zoom byl výškově podobný uživateli



Obrázek 6.13: Import dialog formátu GDF

Kvůli importu a exportu bylo třeba vytvořit dialog pro otevíráni souboru z disku. Jedná se o velice jednoduchý dialog, který obsahuje tlačítka zpět a dopředu, cestu, na které se aktuálně nacházíme, samotné okno se soubory v aktuálním adresáři a tlačítko pro potvrzení výběru.

Obrázek 6.14: Dialog pro výběr souboru

6.6 Implementace graph builderu

Graf lze sestavit mnoha způsoby, proto je tu nástroj zvaný **Graph Builder**, který je určen k „namapování“ zdrojového atributu na kterýkoliv atribut objektu. Tento nástroj také nabízí možnost tyto atributy kombinovat, tudíž ve výsledku je možné nastavit jaký atribut bude kombinovaný s kterým, a jaká operace bude použitá pro tuto kombinaci. Každý atribut má definovaný datový typ, proto nelze kombinovat například textový řetězec s číslem, pokud však není nastaveno, že tento textový řetězec bude převeden na číslo, ovšem to lze jen tehdy, pokud tento textový řetězec je možné převést na číslo. Operace mohou být použité ty, které jsou určené pro kombinaci datových typů těchto dvou atributů, pokud mám dva atributy typu textového řetězce, mohu použít operaci „concat“, která tyto dva řetězce spojí.

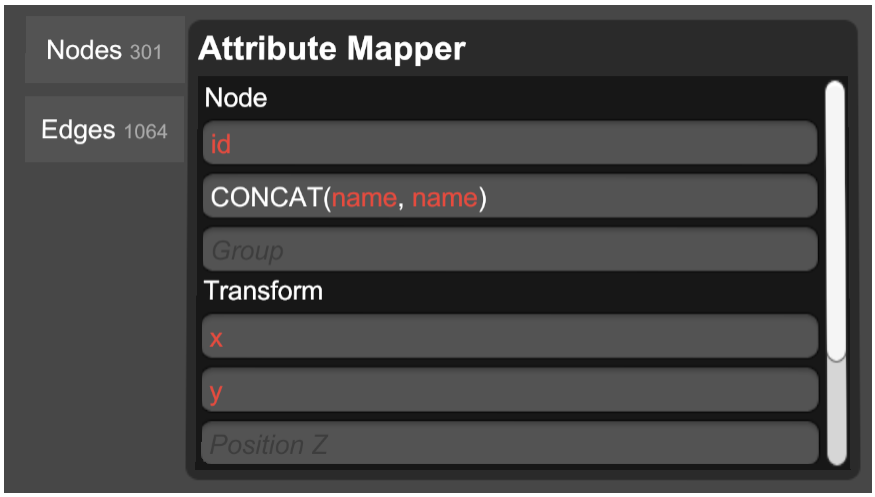
Tento nástroj je použitelný jen pokud byl načtený graf ve formátu, který tento nástroj povoluje atributem *RequiresBuilder*. Třída *GraphModel* (6.2) taktéž obsahuje slovníky *NodePropertyDependencies* a *EdgePropertyDependencies*, kde klíčem je textový řetězec a hodnotou je objekt typu *SelectorList*. Tyto dva slovníky obsahují mapování těchto objektových atributů (např. *RealPosition*) na atributy, které pochází ze zdrojového naimportovaného souboru. Pokaždé, kdy je otevřené *Graph Builder* okno, sbírají se všechny data uzlů a hran, a provádí se určování datových typů těchto dat. Aktuálně existuje jen 5 datových typů pro **Graph Builder** a to jsou *Integer*, *Float*, *Boolean*, *String* a *Color*. Určení datového typu je dle pravého datového typu hodnoty v datech, to znamená, že pokud všechny uzly budou mít v datech klíč *width* s příkladnou hodnotou *22.7*, určený datový typ bude *Float*, avšak pokud alespoň jeden uzel bude mít hodnotu *„nothing“* pod stejným klíčem, určený datový typ bude *String*, protože ne všechny hodnoty pod stejným klíčem mají stejný datový typ a datový typ *String* má vyšší prioritu oproti datovému typu *Integer*. Pokud výsledný datový typ bude *String*, kontrolují se ještě hodnoty tohoto textového řetězce, protože hodnota tohoto textového řetězce může být barva, nebo jiný datový typ. Příkladem může být textový řetězec s hodnotou *„255,255,0,255“*, kde jde o čistě žlutou barvu, a pokud všechny hodnoty všech uzlů pod stejným klíčem budou korektně určeny jako barva, výsledný datový typ bude *Color*, nikoliv *String*. Poté, kdy je určen datový typ klíče je vytvořen objekt typu *SelectorProperty*, který obsahuje název klíče, datový typ, zda-li je obsažen ve všech uzlech nebo hranách a případně jeho minimální a maximální hodnotu.

Priorita datových typů od nejvyšší po nejnižší

- **String**
- **Color**
- **Float**
- **Boolean**
- **Integer**

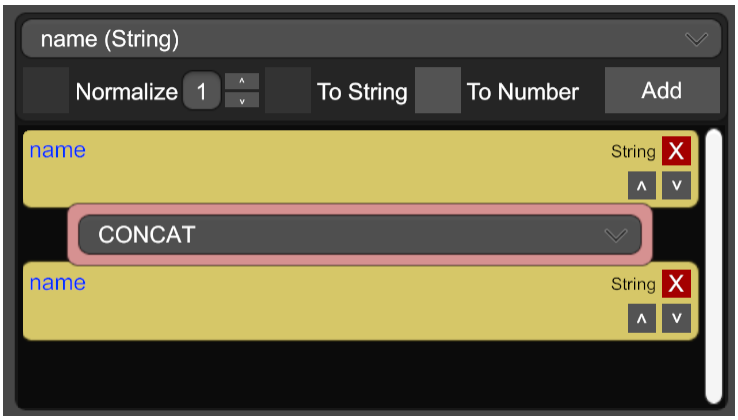
Třída *SelectorList* obsahuje seznam obsahující objekty typu *SelectorObject*, kde *SelectorObject* obsahuje objekt *Property*, *formát*, *Operation*, *AttributeNormalize* a *NormalizeScale*. Jeden objekt typu *SelectorObject* znázorňuje, jaký atribut ze zdrojového souboru si přejeme použít (*SelectorProperty*), jaký výsledný datový typ by měl být (*formát*), jakou operaci použít, pokud seznam obsahuje o záznam navíc (*Operation*), zda-li chceme hodnoty normalizovat (*AttributeNormalize*) a jaký bude násobič normalizovaných hodnot (*NormalizeScale*). Pokud seznam obsahuje více než jeden objekt, všechny objekty by měly mít vyplněnou operaci, která by měla být použitá při kombinování, kromě posledního záznamu, protože neexistuje žádný další objekt, s kterým by mohl hodnotu kombinovat. Může nastat situace, kdy zdrojový atribut je datového typu *Float*, ale hodnota *formát* je nastavená na *Integer*, to znamená, že každá výsledná hodnota musí být převedena do tohoto datového typu a tím pádem číslo ztratí desetinnou čárku.

Operation je objekt rozhraní *ISelectorOperation*, který se stará o operaci, která bude provedena při kombinování aktuálního záznamu s dalším záznamem. Není možné jakkoliv kombinovat záznam s datovým typem *String* se záznamem s datovým typem *Integer*, protože pro tuto kombinaci neexistuje operace a proto je třeba převést jeden z těchto záznamů do stejného datového typu. Existuje výjimka u záznamů typu *Integer* a *Float*, protože oba tyto záznamy jsou numerického typu a proto výsledný datový typ bude *Float*, protože i tyto kombinace se drží pravidel priorit datových typů.



Obrázek 6.15: Graph Builder

Okno na obrázku 6.15 obsahuje dvě záložky pro každý typ objektu v grafu, tudíž uzel a hrana, a jejich aktuální počet. Každá záložka obsahuje objektové atributy, které lze namapovat na zdrojový atribut, nebo kombinaci zdrojových atributů. Toto mapování lze upravit pomocí editoru.



Obrázek 6.16: Editor pro mapování atributů

Editor na obrázku 6.16 zobrazuje výběrový seznam, ve kterém jsou všechny zdrojové atributy připravené k namapování, pod tímto výběrovým seznamem je tlačítko pro přidání atributu a tři zaškrtávací políčka, které jsou určené k změně výsledné hodnoty vybraného atributu, tudíž pokud vybereme ze seznamu atribut číslicového typu, a zaškrtneme políčko *Normalize*, výsledná hodnota by měla být číslo s desetinnou čárkou s hodnotou mezi nulou a hodnotou násobiče normalizace (numerické pole vedle *Normalize*). Výpočet této normalizované hodnoty se provádí pomocí minimální možné hodnoty, maximální možné hodnoty a aktuální hodnoty to celé vynásobané násobičem normalizace. Platí však pravidlo, že pokud minimální možná hodnota a maximální možná hodnota jsou stejné, výsledná hodnota musí být násobič normalizace, protože kdyby byl použit standartní postup, nebylo by možné získat výsledek vzhledem k tomu, že by nastalo dělení nulou.

Výpočet normalizované hodnoty:

$$Value = \begin{cases} \frac{Value - MinValue}{MaxValue - MinValue} \times Scale, & \text{if } MinValue \neq MaxValue \\ Scale, & \text{else} \end{cases} \tag{6.5}$$

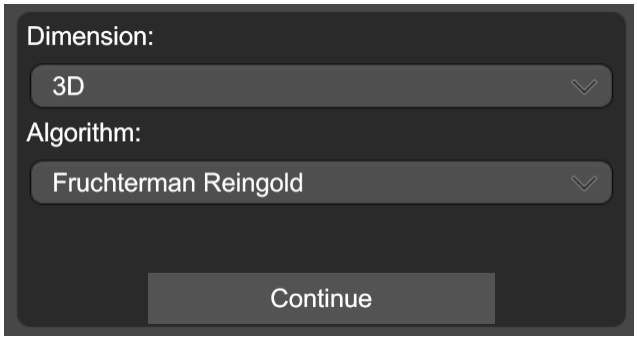
Toto okno také obsahuje seznam všech použitých zdrojových atributů, mezi kterými je vždy vybraná operace, která bude použitá pro zkombinování těchto dvou zdrojových atributů. Každé políčko se zdrojovým atributem taktéž obsahuje informace o datovém typu a jaké další akce jsou s tímto atributem podniknuty (například normalizace). Každé toto poličko lze smazat, nebo přesunou pomocí směrových šipek.

6.7 Implementace výpočtu rozložení

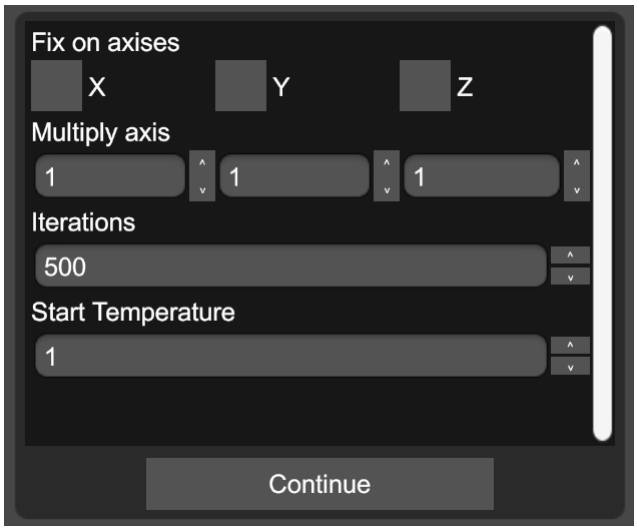
Výpočet rozložení grafu je jedna z nejdůležitějších funkcionalit pro efektivní práci s grafem, vzhledem k tomu, že každý algoritmus je plně konfigurovatelný a také pro každý dvoudimenzionální algoritmus je zde možnost vybrat do jakých dvou os se budou hodnoty naplňovat.

Pro výpočet rozložení se používá externí knihovna zvaná **iGraph** [40], která je psaná v jazyce C++. Průběh výpočtu rozložení funguje tak, že aplikace vygeneruje pole hran, kde každá první hodnota znázorňuje zdrojový uzel a každá druhá hodnota znázorňuje cílový uzel, poté se zavolá metoda pro výpočet rozložení s parametry, které se nastavují v dialogu pro vybraný algoritmus. Tato metoda vytvoří pole vektorových hodnot, které se následně použijí jako pozice uzlů.

Jako je tomu u importu a exportu (6.5), pro vytvoření algoritmu k výpočtu rozložení existuje abstraktní třída *AbstractAlgorithm*, která dědí z abstraktní třídy *AbstractSynchro*. Abstraktní třída *AbstractAlgorithm* obsahuje dvě abstraktní metody *Calculate* a *CreateFields*. Metoda *CreateFields* je volána automaticky hned po tom, kdy bude vytvořen objekt třídy toho jistého algoritmu, a vrací seznam objektů rozhraní *IUIField*, tudíž tato metoda definuje, jaké argumenty daný algoritmus má. Metoda *Calculate* je zavolaná až tehdy, kdy uživatel potvrdí hodnoty vložené do povinných argumentů vytvořené metodou *CreateFields*. Daný algoritmus může najít hodnotu argumentu pomocí generické metody *GetField*, která vrací jakýkoliv objekt typu T dědící z rozhraní *IUIField* a přijímá číselný parametr *Index* jako pozice v seznamu parametrů algoritmu. Všechny již implementované algoritmy funguji na bázi toho, že vytvoří téměř totožný graf v knihovně *iGraph* tím, že bude vytvořeno pole všech hran grafu pomocí metody *GetNativeEdges* a toto pole bude vloženo do vektorového objektu této knihovny, poté bude použita metoda pro výpočet rozložení, která je naimplementovaná již v této knihovně a budou použitý parametry, které byly již definovány v metodě *CreateFields* a přečteny pomocí *GetField*. Po dokončení výpočtu se vytvoří seznam Unity *Vector3* objektů, který je vyplněn pomocí metod této knihovny a poslán jako parametr metody *Adjust*, která nastaví reálnou pozici všem uzlům dle tohoto seznamu. Po dokončení implementace musí být algoritmus přidán do slovníku *Layout2dAlgorithms* nebo *Layout3dAlgorithms* v komponentě *GraphManager* dle cílového typu algoritmu, kde klíčem bude název algoritmu a hodnota bude reference typu třídy algoritmu.



Obrázek 6.17: Výběr algoritmu pro výpočet rozložení



Obrázek 6.18: Parametry algoritmu pro výpočet rozložení

6.8 Implementace klávesnice

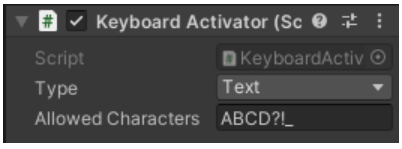
V jistých částech aplikace jsou očekávána nějaká vstupní data uživatele, ale ve virtuální realitě je náročné používat klávesnici, kterou máme u počítače. Proto je třeba zaplnit i tento nedostatek pomocí virtuální klávesnice. Ovšem SteamVR má vlastní klávesnici, ale bohužel v Unity nelze použít, z tohoto důvodu bylo třeba vymyslet vlastní virtuální klávesnici, což má i své výhody, a jedna z nich je tzv. „ušití na míru“, to znamená, že tato klávesnice je užitečná ve všech oblastech aplikace, ve kterých je klávesnice požadována nebo volitelná.

Klávesnice má tři módy:

1. Textový mód
2. Numerický mód
3. Výrazový mód (Regexr mód)

Textový mód je postaven přibližně tak, aby odpovídal rozložení v mobilních klávesnicích, tudíž lze přepínat mezi jednoduchými textovými znaky (písmeny) a ostatními znaky. Numerický mód je postaven tak, aby byl užitečný nejen na vkládání číslíc ale i jejich výpočet. A poslední výrazový mód je určen k vytváření regulárních výrazů, proto tato klávesnice obsahuje všechny potřebné znaky k vytvoření jakéhokoliv výrazu.

Komponenta, která se stará o funkcionalitu klávesnice je *KeyboardController*, která dědí z abstraktní třídy *AbstractController*. Každé textové pole, které je určené k úpravě pomocí klávesnice musí mít komponentu *KeyboardActivator*, která zajišťuje, že po stisknutí tlačítka spouště (Trigger) s ukazatelem mířeným na určené textové pole otevře klávesnici. *KeyboardActivator* má nadefinované dvě konfigurační proměnné *Type* a *AllowedCharacters*. *Type* je jedna z hodnot enumerátoru *KeyboardType*, která má tyto tři možnosti: *Text*, *Number*, *Regexr*. Zajišťuje tak, v jakém módu bude klávesnice otevřena, a *AllowedCharacters* je textový řetězec, který nastavuje, jaké znaky mohou být povolené ve výsledném řetězci z klávesnice. Pokud je *AllowedCharacters* prázdný řetězec, klávesnice není ničím omezena. Hned po otevření klávesnice je okno, ve kterém se upravovaný textový řetězec nachází posunuje dozadu a na jeho původní místo je zobrazeno okno klávesnice. V případě uzavření klávesnice je cílené okno opět posunu na svoji původní pozici před otevření okna klávesnice.

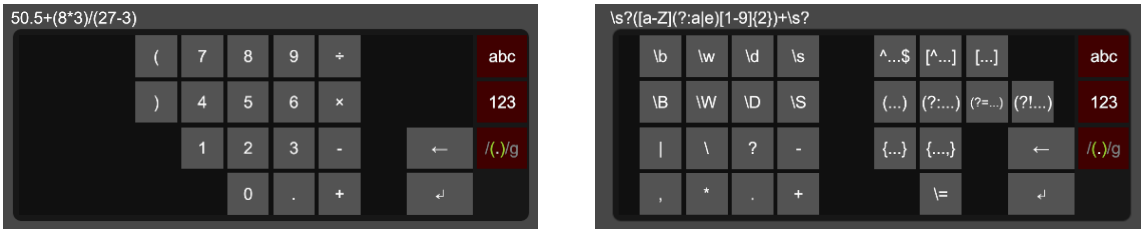


Obrázek 6.19: KeyboardActivator komponenta v inspektoru

Numerický mód klávesnice taktéž nabízí možnost kalkulačky díky tlačítkům pro číselné operace, avšak platí pravidlo, že pokud klávesnice nebyla otevřena v numerickém módu, tudíž i textové pole

není numerického typu, nelze použít kalkulačku a proto není výsledný vzorec zkontrolován. Kontrola vzorce se provádí pomocí metody *EvaluateNumber*, ve které je využita třída *DataTable* [41] a její metoda *Compute*, která vypočítá zadaný vzorec a v případě chybného vzorce změní barvu vzorce na světle červenou.

Klávesnici lze ovladat pomocí ukazatele pravého ovladače nebo Joysticku / Trackpadu obou ovladačů tak, jak bývá zvykem u původní SteamVR klávesnice.



(a) Numerická (b) Výrazová

Obrázek 6.20: Numerická a Výrazová virtuální klávesnice

6.9 Implementace osy

Vizualizace osy je volitelná možnost, to znamená, že uživatel tuto možnost může vypnout nebo zapnout dle svých vlastních preferencí. Osy lze nastavovat i po částech, tudíž uživatel nemusí vypnout všechny osy, ale jenom některé, nebo vypnout zobrazení textu (číslic) na některých osách.

Osa Z a osa Y lze zaměnit taktéž dle konfigurace pod klíčem *ReverseHeightAxises* nebo v hlavním menu. Původní výšková osa je Y, ale pokud jsou osy Y a Z zaměněné, tak i vypočtená aktuální pozice uzlu bude takhle pozměněná, protože v Unity nelze definovat, jaká osa bude ta výšková, a proto je třeba toto pozměnit v aplikaci, proto hodnota Y vektoru bude hodnota Z z dat. O osy se stará komponenta *GraphAxisController*, která automaticky skrývá a odkrývá části os dle konfiguračních atributů podle klíčů *ShowXAxis*, *ShowYAxis*, *ShowZAxis*. Lze taktéž skrýt názvy os a jejich číselníky pomocí konfiguračních atributů *ShowXName*, *ShowYName*, *ShowZName*, *ShowXNumbers*, *ShowYNumbers* a *ShowZNumbers*. Všechny tyto konfigurace jsou původně nastavené na hodnotu *True*, ale lze je vypnout v hlavním menu aplikace nebo úpravou konfiguračního souboru. Obnova číselníku osy se provádí jen tehdy, pokud nastane změna v grafu, příkladem je změna zoomu. Tato úprava byla vyžadována vzhledem k tomu, že je postup vytváření číselníku náročný, a jak bylo v předešlých verzích práce, tento číselník se aktualizoval kdykoliv kdy nastal *FixedUpdate*. *FixedUpdate* je Unity metoda, která nastane kdykoliv, kdy se přepočítává fyzika, tudíž 50 krát za sekundu. Toto řešení nebylo vhodné a proto aktualizace je volána jen tehdy, kdy je to nutné, avšak s tímto řešením je třeba kontrolovat, zda-li všechny možnosti, které by mohly vyžadovat tuto aktualizaci jsou doplněny tímto voláním. Metoda *DoFullUpdate*, která se stará o již zmíněnou aktualizaci vykonává až pět kroků:

1. Aktualizace viditelnosti všech os
2. Vytváření a vykreslení číslic
3. Aktualizace barev os
4. Rotace všech číslic na kameru

Vytváření a vykreslení číslic funguje tak, že první zkontroluje, zda-li je aktuálně na scéně načtený nějaký graf, pokud ne, číselník bude vyprázdněn. V opačním případě je třeba zjistit extrémy grafu z pohledu aktuálních pozic, tudíž nejnižší bod a nejvyšší bod, oba tyto body budou převedeny do absolutních hodnot a bude zjištěno, jak velká by měla být každá osa pro kladná i záporná čísla. Potom bylo třeba vypočítat aktuální a reálnou velikost grafu, díky těmto hodnotám lze vypočítat velikosti jednoho metru v reálných souřadnicích. Číslice pro číselník bude vytvořena každý půl metr, tak jak je nastaveno v konfiguračním souboru pod klíčem *AxisNumberStep*, kde je nastavena hodnota *0.5*. Počet číslic pro každou osu je vypočítaná velikost osy vydělena hodnotou *0.5*, tedy nastaveným krokem na jeden metr (*AxisNumberStep*) a potom vynásobena hodnotou *2*, protože osa

má kladnou i zápornou část.

$$\begin{aligned} Size &= \begin{cases} |Minimum_{xyz}|, & \text{if } |Minimum_{xyz}| \geq |Maximum_{xyz}| \\ |Maximum_{xyz}|, & \text{else} \end{cases} \\ NumberCount_{xyz} &= \frac{Size_{xyz}}{AxisNumberStep} \times 2 \end{aligned} \tag{6.6}$$

Pokud je počet číslic pro každou osu změněn, je třeba odebrat nebo přidat objekty číslic, které se na scéně budou vyskytovat. Pro každé nové číslice bylo potřeba vypočítat jeho pozici, a ta byla vypočítána pomocí offsetu grafu vynásobeného zoomem grafu, kladnou nebo zápornou jedničkou, podle toho na jakou stranu má být číslice vypočítana, a krokem na jeden metr.

$$Position = (Index + 1) \times AxisNumberStep \times Positive + Offset \times Zoom \tag{6.7}$$

Hodnota těchto číslic je vypočítána, až na to, že je třeba počítat i s měřítkem na jeden metr:

$$Value = (Index + 1) \times AxisNumberStep \times Meter \times Positive + Offset \tag{6.8}$$

Barvy číslic a os lze měnit v konfiguračním souboru podle klíčů *XAxisColor*, *YAxisColor*, *ZAxisColor*. Původní barvy se řídí podle schématu RGB, tudíž osa X má světle červenou barvu (přesně *255, 96, 96*), osa Y má světle zelenou barvu (*96, 255, 96*) a osa Z má světle modrou barvu (*96, 96, 255*).

6.10 Načítaní konfigurace ze souboru

Konfigurace tvoří aplikaci přizpůsobitelnou. Proto i tato funkcionalita byla přidána do této aplikace. Konfigurace je i jinak zvána jako uživatelské preference. Většina míst aplikace lze přizpůsobit dle této konfigurace. Konfigurační soubor se nachází v pracovní složce aplikace pod názvem *config.xml*. Jak již název napovídá, tento soubor je ve formátu XML [42]. XML formát byl vybrán kvůli své jednoduchosti, a rychlé implementaci.

Uživatelská konfigurace se nachází ve statické třídě *UserConfiguration*, kde se nachází statické proměnné. Každá tato proměnná má shodný název s klíčem v konfiguračním souboru. O serializaci této třídy se stará statická třída *ApplicationConfiguration*, kde jsou metody *LoadConfiguration* a *SaveConfiguration*. *LoadConfiguration* je volán hned po spuštění aplikace, a *SaveConfiguration* je volán po ukončení aplikace. K ukládání a načítání je třeba použít tzv. Reflexi. Reflexe je způsob, jak prozkoumávat třídy, metody, proměnné, atd. [43]. V tomhle případě je Reflexe užitečná k tomu, abych zjistil všechny statické proměnné z třídy *UserConfiguration* a pokusil se názvy těchto proměnných vyhledat v konfiguračním souboru nebo ten konfigurační soubor vytvořit. K načtení a uložení byla využita třída *XDocument* [44]. Struktura konfiguračního souboru je následující:

```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <attribute name="ColorizeEdges">True</attribute>
  <attribute name="EdgeStartColor">8,32,64,255</attribute>
  <attribute name="EdgeEndColor">8,32,64,255</attribute>
  <attribute name="ColorDistance">1.0,20.0</attribute>
  ...
</config>
```

Listing 6.1: Struktůra konfiguračního souboru

Kde *config* je kořen XML dokumentu a *attribute* je objekt tvořící konfigurační hodnotu pod klíčem *name* a hodnotou uvnitř objektu. Hodnoty mohou být různého typu, ale metoda, která se stará o načítání konfigurace přijímá jenom typy *Int32*(integer), *Single*(float), *Boolean*(bool), *Color* a *Vector2*. Protože jiné typy nebyly třeba ukládat.

6.11 Implementace vyhledávání v grafu

Vyhledávání je jednou z dalších pokročilejších funkcionalit aplikace. Vyhledávat uzly nebo hrany manuálním způsobem může být časově velice náročné, a proto je tu možnost vyhledávat tyto objekty dle nějakého filtru. Tento filtr lze nastavit na dva módy:

1. Přímý výraz
2. Regulární výraz

Přímý výraz vyhledává v datech jen ty hodnoty, které v sobě obsahují daný text (např. Multi**function**al). Ovšem nejužitečnější je vyhledávání dle regulárního výrazu, vzhledem k tomu, že regulární výrazy nám dávají obrovské možnosti jak data filtrovat.

Obrázek 6.21: Vyhledávání

Na obrázku 6.21 lze vidět zaškrtávací pole **Regular Expression**, který přepíná mezi přímým a regulárním módem vyhledávání. Pod tímto zaškrtávátkem je textové pole, do kterého lze zapsat potřebný výraz. Dále k omezení zobrazování mnoha výsledků je zavedeno strankování, díky kterému lze přepínat mezi stránkami, které obsahují maximálně až sto výsledků. Přesný počet aktuálně zobrazených výsledků a celkových výsledků je možné vidět v horní části vyhledávácího okna vedle podpisu **Results**. Vyhledané záznamy jsou vypsány ve výsledkovém podokně ve formátu typu objektu, jeho identifikačního čísla a názvu klíče s hodnotou oddělenou znakem rovnítka. Každý vyhledaný záznam taktéž obsahuje tlačítko **Select**, které nalezený objekt označí v grafu a zároveň otevře detailní okno tohoto objektu.

Vyhledávání začne až tehdy, kdy do políčka **Search Expression** bude vložen řetězec, ten spustí listener, který volá metodu *DataSearch_ExpressionChanged* ve třídě *CanvasMethodProcessor*. Princip vyhledávání je takový, že se prvně vytvoří seznam všech entit grafu (uzly a hrany), a tento seznam je následně seřazen dle *Id*. Je taktéž vytvořen prázdný seznam, který bude sloužit jako

seznam výsledků. Pro každou entitu bude procházen každý záznam v datovém slovníku a vytvoří se těxtový řetězec podle šablony *<Klíč>=<Hodnota>*. Pro tento vytvořený textový řetězec bude zkontrolováno, zda-li obsahuje text nebo je vyhodnocen jako pozitivní dle regulárního výrazu, který se nachází v poli **Regular Expression**. Pokud bude nalezena shoda, záznam bude vložen do výsledkového seznamu. Až když je vyhledávání dokončeno, pro káždý záznam ze seznamu výsledků bude vytvořeno pole, které bude obsahovat typ entity, jeho *Id*, a vytvořený textový řetězec určený k vyhledání podobnosti, ve kterém bude vyznačená část, díky které bylo vyhledání uspěšné. Celkový počet nalezených záznamů bude zapsát do textového pole **Results**.

6.12 Implementace načítání zásuvných modulů

Aplikace, která se věnuje tomuto tématu nemůže být nikdy pro každého dokonalá, protože každá jiná aplikace je něčím výjimečná, a není možné vytvořit aplikaci s každou funkcionalitou jiných aplikací, proto existuje koncept zásuvných modulů. Tento koncept je založen na tom, že zásuvný modul je softwarová komponenta, která přidává specificke funkcionality do již existující aplikace za účelem tuto aplikaci rozšířit. Proto i tato aplikace podporuje vytváření těchto rozšiřujících komponent a tím aplikaci dělat užitečnější. Stejným důvodem bylo potřeba vytvořit rozhraní pro programování bez toho, aby byl prozrazen zdrojový kód aplikace. Kvůli tomu bylo třeba vytvořit tzv. „Zrcadlené třídy“, což jsou třídy s *get* a *set* vlastnostmi se stejnými jmény jako jsou v zrcadlené třídě, a tyto vlastnosti se dotazují rozhraní na jejich aktuální hodnotu. Příkladem může být třída pro uzel *GraphNode*, protože je samozřejmé, že pokud vývojář tvoří nějaký zásuvný modul do aplikace pracující s grafy, určitě bude pracovat s uzly nějakého grafu. Tudíž pro třídu *GraphNode* byla vytvořena zrcadlena třída se jménem *Node*. Třída *GraphNode* v sobě ukládá informace o jistém uzlu, a třída *Node* se ptá aplikace pomocí rozhraní na tyto informace. Ve výsledku když je volána *get* metoda vlastnosti *Id*, je taktéž volána metoda v rozhraní, která na straně aplikace převede objekt *Node* na objekt *GraphNode*, získá hodnotu *Id* a vrátí. Stejná funkcionalita platí i pro objekty grafů a hran.

Další užitečnou možností je vytváření další podpory pro formáty importu a exportu, algoritmů pro výpočet rozložení a operací v Graph Builderu (6.6). Každý zasuvný modul musí dědit ze třídy *Plugin* a být označen atributem *GraphViewPlugin*, který musí mít vyplněné informace *Guid*, *Name*, *Author* a *Version*. *Guid* je unikátní id zásuvného modulu, příkladem může být „com.domain.name“. *Name* je název zásuvného modulu, *Author* je jméno autora a *Version* je verze zásuvného modulu. Jako další volitelná informace je *Dependencies*, což je seznam unikátních id zásuvných modulů, na kterém je tento zásuvný modul závislý. Seznam závislosti zaručí, že načtení tohoto zásuvného modulu proběhne až tehdy, kdy jsou načteny všechny závislosti. Problém může naskytnout tehdy, pokud jistá závislost není načtená, nebo neexistuje, potom nedojde k načtení závislého zásuvného modulu. O načítání a odnačítání zásuvných modulů se stará statická třída *PluginController*, která se taktéž stará o převod zrcadlených tříd na reálné.

Zásuvné moduly jsou uloženy ve složce **Plugins** v pracovním adresáři aplikace. Každý zásuvný modul je soubor s přeponou „.dll“. Je spuštěn cyklus, který najde všechny tyto soubory a pokusí se je načíst tím, že zjistí, jestli existuje nějaká třída s atributem *GraphViewPlugin*, a pokud taková třída existuje, zjišťuje se, zda-li jsou splněná podmínka načtených závislostí. Pokud je tato podmínka splněná, zásuvný modul je načten a uložen do seznamu načtených zásuvných modulů, ale pokud není podmínka splněná, je zásuvný modul vložen do čekající fronty. Až když cyklus doběhne, je spuštěn další cyklus, který prochází frontu čekacích zásuvných modulů a provádí se pokus o načtení těchto modulů tím, že se kontroluje podmínka, zda-li jsou závislosti načtené nebo jsou taktéž v čekající frontě. V případě čekající závislosti se první provede pokus o načtení této závislosti a až pak závislého čekajícího zásuvného modulu. V případě oboustranných závislostí není možné, aby

jediný z těchto zásuvných modulů bych načten. Poté kdy modul úspěšně splní podmínku závislostí ho lze načíst tak, že se zavolá metoda *OnLoad* ve třídě modulu, a jsou zjištěny další třídy, které slouží k rozšíření aplikace, jako je například podpora pro další formát importu a exportu.

Všechny zásuvné moduly jsou odnačteny v moment, kdy se provádí ukončení aplikace. Odnačtení je provedeno ve stejném pořadí v jakém byly moduly načteny. Odnačítací metoda zásuvného modulu je *OnUnLoad*.

Kapitola 7

Způsoby optimalizace

Obecným pojmem optimalizace se ve 3D počítačové grafice označuje proces, který má za cíl snížit časovou, nebo paměťovou náročnost pro vykreslení snímku. Tyto optimalizace je třeba zavést, aby bylo možno na scéně vykreslit více objektů dle potřeb samotného uživatele. Optimalizace taktéž neznamená vypnutí vlastností nebo snížení detailů aplikace. Ideálním řešením je scénu i chod aplikace optimalizovat tak, aby uživatel nepřišel o žádné užitečné vlastnosti aplikace a vzhled scény byl zachován, ale i přesto došlo k navýšení snímkové frekvence. Mnohdy lze tohoto dosáhnout správným navržením aplikace a dodržením jistých stanovených limitů.

Aplikace by se měla tvořit s kontinuálním přehledem o náročnosti scény a kódu. Ve finále je tvorba aplikace mnohem snažší s ohledem na stanované limity, než se optimalizacemi zabývat v poslední části vývoje, kdy bude daleko náročnější modifikovat její části, díky celkové větší komplexnosti aplikace [45].

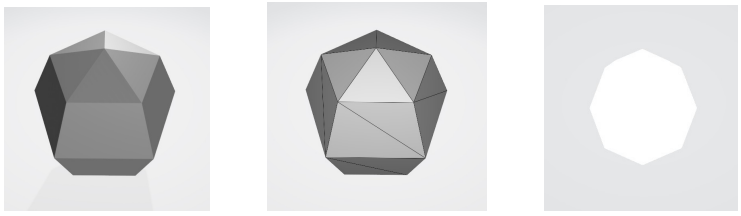
Optimalizace jsou ještě důležitější v případě vývoje pro virtuální realitu. Vzhledem ke kombinaci vysoké snímkové frekvenci (90Hz) a dvou OLED panelů v headsetu s vysokým rozlišením je nutno scénu renderovat dvakrát (pro každé oko), díky tomu hraje optimalizace klíčovou roli. Této problematice byl věnován patřičný prostor.

7.1 Rozlišení

Jedna z nejjednodušších optimalizačních metod se dá považovat rozlišení vykreslování, které se velice značně podepisuje na snímkové frekvenci. Tato možnost je dostupná i ve virtuální realitě. Obrovskou nevýhodou zmenšení rozlišení je znečnější rozmazanost výsledného obrazu.

7.2 Geometrie uzlů

Hlavní součástí aplikace je vizualizace nám vytvořeného grafu, a ten se skládá z velké množiny uzlů. Tuto množinu je potřeba efektivně vykreslit aniž by tato akce nesla kritické následky na snímkovou frekvenci aplikace, která by zapříčinila nepříznivé podmínky pro práci. Proto je vhodné pro uzel využit ten nejméně náročný model (30 trojúhelníků).



(a) (b) (c)

Obrázek 7.1: Model uzlu

Nevýhodou tohoto nedetailního modelu je vzhled aplikovaných shaderů na jeho materiál. Původně byl použitý shader na obrys modelu jinou barvou, než je barva materiálu, pokud uživatel tento uzel označí, ale vzhledem k nízkému počtu trojúhelníku je tento obrys nekvalitní. Proto bylo vymyšleno jiné řešení, a to je přidání označující barvy s 50% průhledností k barvě materiálu uzlu.

7.3 Redukce draw callů

Jednou z velice efektivních metod optimalizace je redukce draw callů. Draw call je instrukce, kterou vytváří CPU a předává GPU ke zpracování. Jedno toto volání obsahuje informace o tom jaký objekt je potřeba vykreslit (mesh), jaké textury tento objekt má a jeho ostatní informace (např. osvětlení). Problém však nastává tehdy, pokud máme na scéně mnoho objektů, které je třeba vykreslit pro každý jeden snímek. Čím více objektů se vyskytuje na scéně, tím více času je potřeba pro přípravu těchto volání, viz. Unity Profiler znázorňující detailní informace o aktuálně vykresleném snímku. Proto existuje koncept materiálů [46]. Materiál je datová struktůra obsahující informace o tom, jak vykreslit objekt, a jaký shader k tomu použít. Díky tomuto konceptu je snadné shromáždit všechny objekty, které využívají tento materiál a vykreslit je najednou, tzv. Group batching.

Obrázek 7.2: Unity Profiler

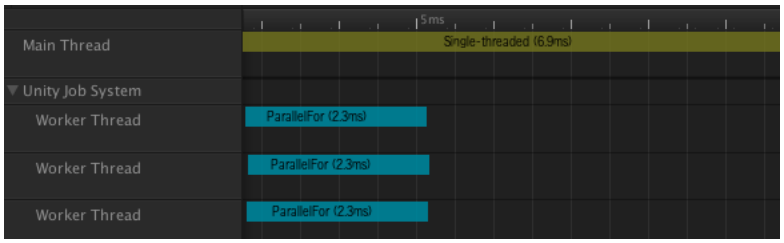
7.4 Data-Oriented Technology Stack

Data-Oriented Technology Stack (dále jen DOTS) se skládá ze tří hlavních komponent:

- C# Job System
- Entity Component System
- Burst Compiler

7.4.1 C# Job System

C# Job System využívá konceptu multi-threadingu[1], který samostatně řídí všechna dostupná vlákna a přiřazuje jim úkoly [47].

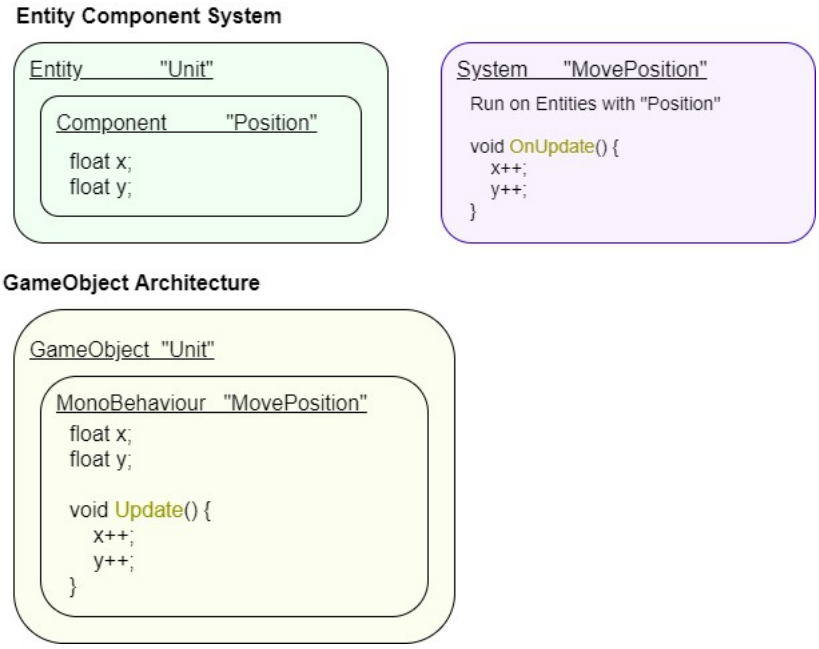


Obrázek 7.3: C# Job System

[1] **Multi-threading** - Využití více vláken procesoru

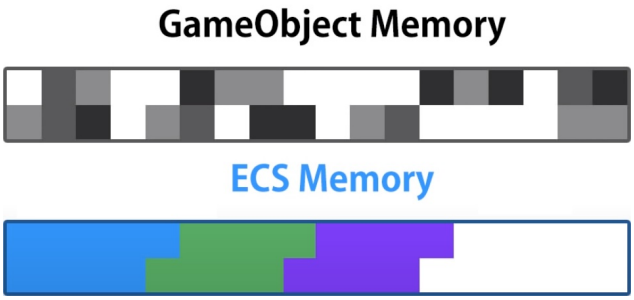
7.4.2 Entity Component System (ECS)

ECS vytváří nový typ systému, jak udržovat objekty na scéně. Původní architektura Unity se skládá z GameObject instancí, které si udržují MonoBehaviour komponenty, avšak nová DOTS architektura se skládá z entit, které odkazují na komponenty udržující data, které jsou zpracovávány systémy.



Obrázek 7.4: Unity Entity Component System

Hlavním rozdílem je správa paměti pro entity, jelikož entity jsou uspořádany v paměti na sobě, tak aby CPU nemuselo přeskakovat a hledat, kde se nachází další objekt.

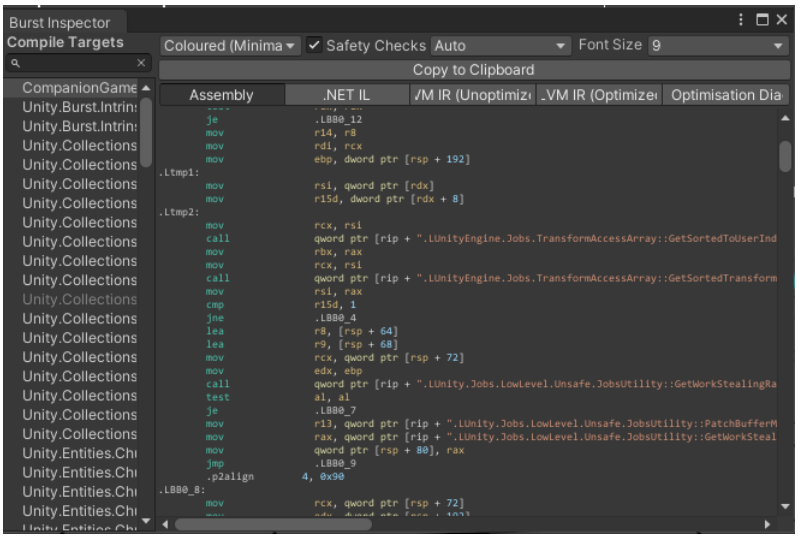


Obrázek 7.5: ECS Paměť

Výhodou využití ECS nastává, pokud je využit i s job systémem, který je převážně navržený kvůli tomuto účelu [48].

7.4.3 Burst Compiler

Burst Compiler zkompiluje C# kód do velice optimalizovaného strojového kódu, převážně velice výhodné v optimalizaci job systému [49]. Za pomocí tohoto systému lze vybudovat nástroj schopný zvládnout velký nátlak objektů na scéně s minimálním rozdílem na snímkové frekvenci. Proto bude využit k vizualizaci grafu [50].



Obrázek 7.6: Burst Inspector

Burst inspector dovoluje vidět všechny zkompilováné třídy, které jsou označeny **BurstCompile** atributem v assembly jazyce, nebo-li „assembleru“.

Kapitola 8

Testování aplikace

8.1 Testování

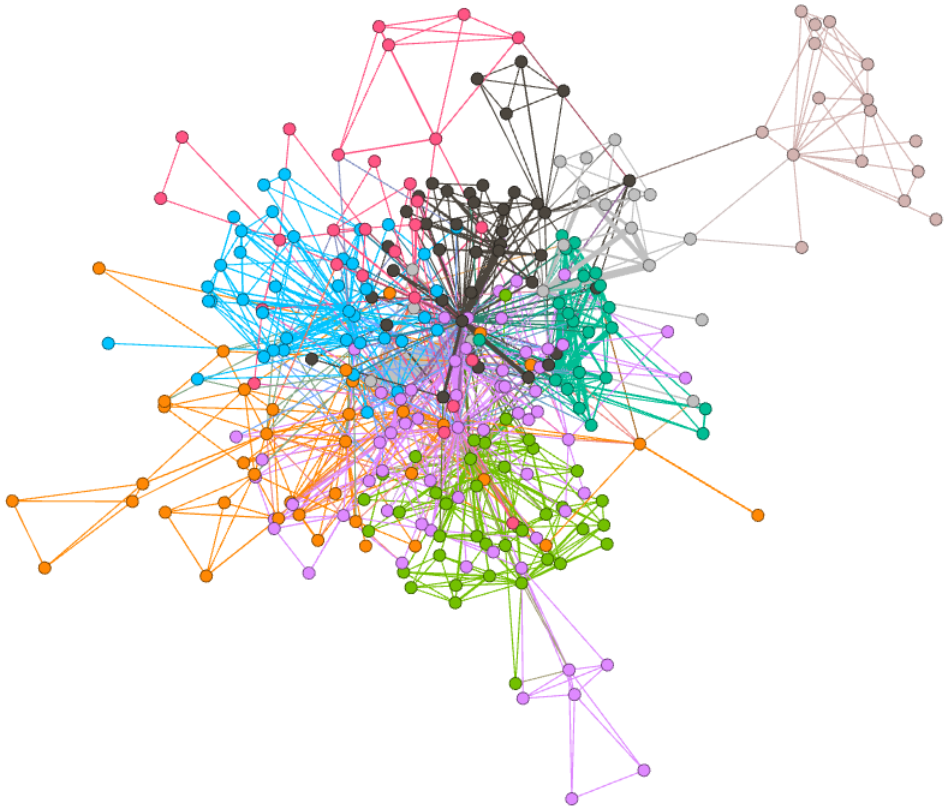
V první fázi vývoje aplikace bylo třeba vytvořit snadný import dvou CSV souborů určené k informacím o uzlech a hranách, díky kterým byl zkonstruován jednoduchý graf. Pro tuto fázi vývoje stačil jednoduchý dialog.V druhé fázi vývoje bylo třeba implementovat renderování scény do virtuální reality, proto k testování posloužil školní VR headset **HTC Vive**, který běží na platformě **SteamVR**. Další fáze aplikace implementovaly vždy novou funkcionalitu a proto bylo třeba pro každou novou funkcionalitu vytvořit testovácí scénář, který byl automaticky spuštěn a plnil funkci jako virtuální uživatel. Zároveň bylo třeba funkcionalitu otestovat i samostatně. Díky již vytvořeným testovacím scénářům jsem byl schopen odhalit chyby, které vznikly při vývoji aktuální funkcionality, dalších funkcionalit nebo závažných změn v aplikaci, které především nastaly při optimalizaci a implementaci zásuvných modulů. V pozdější fázi aplikace byl využit i mimoškolní VR header **Oculus Quest 2**, který posloužil jako další testovací nástroj kompatibility. Díky použití i jiného VR headsetu jsem došel k tomu, že každý headset, který běží pod platformou SteamVR je kompatibilní s aplikaci. V první polovině vývoje byl vývoj průběžně testován v laboratoři, ale vzhledem k nouzovému stavu nebylo možné druhou polovinu vývoje testovat v laboratoři a tak ukázat aplikaci lidem, kteří často pracují s vizualizačními aplikacemi, a tím získat zpětnou vazbu k zachycení těch správných a užitečných zvyků, díky kterým by aplikace dostala přesnější směr a tak zaručila, že noví uživatelé by se naučili tuto aplikace ovladat mnohem snadněji.

CPU	RAM	GPU	VRAM
AMD Ryzen 7 1700	48GB DDR4	AMD Radeon RX 570	4GB GDDR5

Tabulka 8.1: Použitý hardware pro vývoj a testování

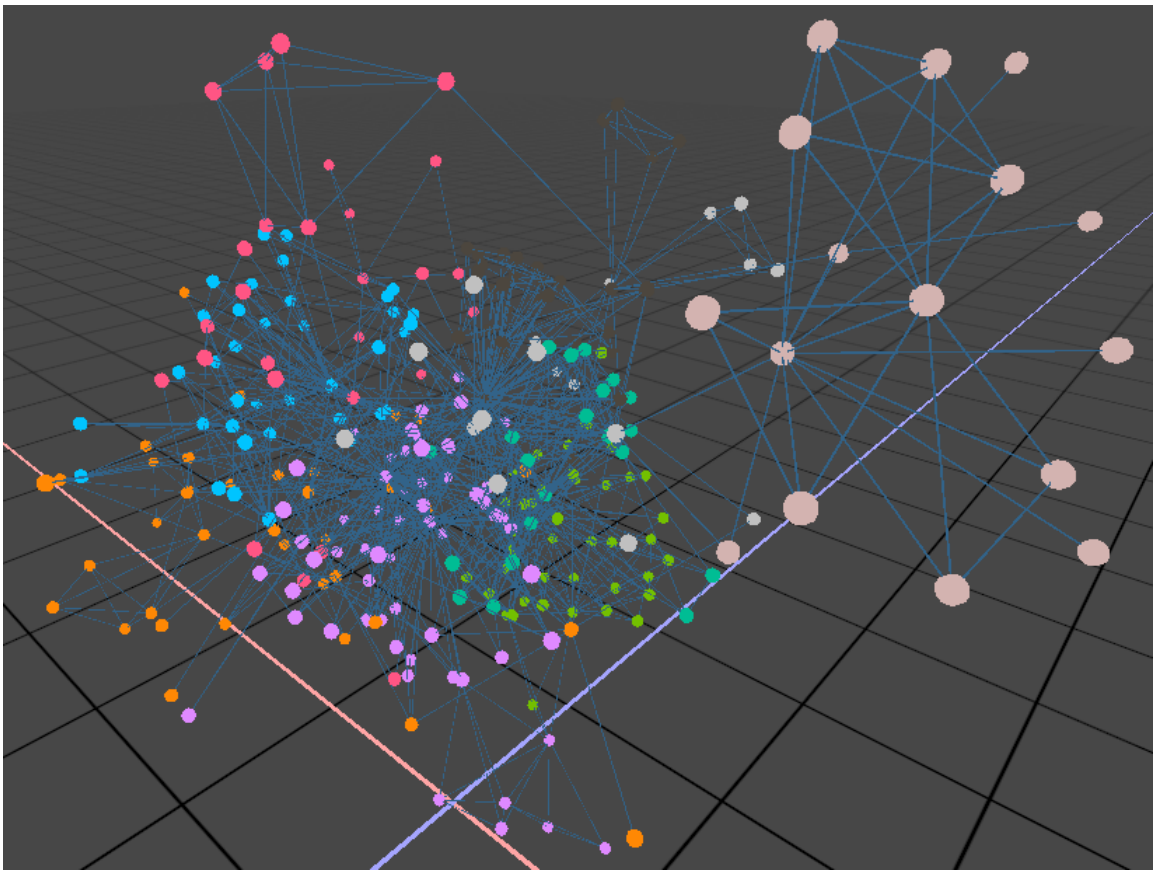
8.1.1 Porovnání grafu v jiné aplikaci

K porovnání byla využívána aplikace Gephi [6]. Graf, který je použit v porovnávání byl načten a změněn v projektové aplikaci tak, aby bylo co nejlépe vidět využití Fruchterman-Reingold algoritmu ve třídimenzionálním prostoru. Tento vytvořený graf byl poté vyexportován ve formátu GDF [36] a načten v aplikaci Gephi.



Obrázek 8.1: 3D graf v Gephi

V projektové aplikaci nebylo provedeno nic jiného, než načtení, spuštění algoritmu pro výpočet rozložení a samotného exportu. Pro zlepšení vizualizace grafu lze použit funkcionalitu, která mění barvu uzlu dle vzdálenosti kamery od uzlu. Navíc lze jednoduše měnit tloušťka hran, velikost uzlu, atd. Ale v tomhle případě porovnání je ukázano, co lze vytvořit během pár sekund užívání.



Obrázek 8.2: 3D graf v projektové aplikaci

Pro správnou orientaci jsou vizualizované osy X (červená) a Z (modrá). Y osa je v tomto případě výšková kvůli důvěryhodnějšímu zobrazení v Gephi po exportu.

Kapitola 9

Závěr

Cílem této práce bylo nastudovat komplexní sítě a vytvořit tak aplikaci schopnou tyto sítě vizualizovat ve virtuální realitě za pomocí herního enginu Unity.

V teoretické části se práce zabývá cíli a možnostmi, díky kterým práce získala jistý směr. Hlavní myšlenkou byla vizualizace komplexních sítí ve virtuální realitě. Podařilo se vytvořit tuto vizualizaci ve velice optimalizovaném stavu, díky systému DOTS, které herní engine Unity nabízí. Navíc byla vytvořena podpora pro import grafu s možnosti nastavení různých atributů ze zdrojového souboru na atributy entit grafu a tím mít potenciální možnost pro podporu načítání grafů vytvořených jinými nastroji k vizualizaci. Taktéž byla vytvořena podpora pro export grafu, výpočet rozložení včetně seznamu několika známých algoritmů a možnosti přidat další, díky podpoře zásuvných modulů. A nakonec lze v grafu i vyhledávat pomocí regulárního výrazů.

V praktické části se práce zabývá popisu vývoje aplikace pro virtuální realitu v Unity, implementace různých částí aplikace a následné optimalizace.

Největší problém při vývoji aplikace dělalo udržení stability počtu snímku za sekundu při různě komplexních grafech, vzhledem k tomu, že renderovat scénu pro obě oči VR headsetu a obrazovku může být pro hardware obtížné. V průběhu této optimalizace bylo potřeba změnit mnoho částí aplikace, a kvůli tomu byl zaveden systém automatických testů, díky kterým jsem odhalil chyby, které vznikly při změnách aplikace.

Práce s grafy a celkový proces vývoje aplikace byl místně fascinující, avšak časově náročný kvůli udržení stabilního a bezproblémového běhu aplikace. Unity engine se ukázal jako ideální nástroj ve vizualizaci komplexních sítí díky možnosti využít hardware až na 100%, a tak docílit plynulého běhu aplikace i při práci s náročnějšími grafy. Vzhledem k získaným zkušenostem ohledně práce s grafy a možnostmi týkající se jejich vizualizace, bych rád pokračoval na tomto projektu i v rámci navázujícího magisterského studia vylepšením celkové práce s grafy, a funkcionalitou, která by dovolila synchronizovaně pracovat nad jedním grafem ve více lidech.

Literatura

1. *Unity Engine* [online]. 2020 [cit. 2020-12-30]. Dostupné z: https://unity.com/.
2. *What is Virtual Reality?* 2019. Dostupné také z: https://www.marxentlabs.com/what-is-virtual-reality/.
3. *VIVE™| Buy VIVE Hardware* [online]. HTC Corporation, 2019 [cit. 2020-12-30]. Dostupné z: https://www.vive.com/eu/product/.
4. *Oculus Quest 2*. 2020. Dostupné také z: https://www.oculus.com/quest-2/.
5. *Virtualitics*. 2021. Dostupné také z: https://virtualitics.com/.
6. *Gephi* [online]. 2017 [cit. 2020-12-31]. Dostupné z: https://gephi.org/about/.
7. *Neo4j Bloom* [online]. 2020 [cit. 2021-01-01]. Dostupné z: https://neo4j.com/product/bloom/.
8. *ReactJS*. 2021. Dostupné také z: https://reactjs.org/.
9. *A Gentle Introduction To Graph Theory* [online]. 2017 [cit. 2021-01-03]. Dostupné z: https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8.
10. *Graph Data Structure And Algorithms*. 2018. Dostupné také z: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/.
11. *The web as a directed graph*. 2020. Dostupné také z: https://computersciencewiki.org/index.php/The_web_as_a_directed_graph.
12. ALBERT, R.; BARABÁSI, A.-L. *Statistical mechanics of complex networks*. 2002.
13. NEWMAN, Mark. *Networks: An Introduction*. 2010.
14. COHEN, Reuven; HAVLIN, Shlomo. *Complex Networks: Structure, Robustness and Function*. 2010.
15. *COMPLEX NETWORKS*. Dostupné také z: https://physics-complex-systems.fr/complex-networks.html.
16. *INTRODUCTION TO COMPLEX NETWORK ANALYSIS*. Dostupné také z: https://medium.com/analytics-vidhya/introduction-to-complex-network-analysis-15b50947a794.

17. *Unreal Engine*. 2021. Dostupné také z: https://www.unrealengine.com/.

18. *Game Maker: Studio*. 2021. Dostupné také z: https://www.yoyogames.com/en/gamemaker.

19. *Godot*. 2021. Dostupné také z: https://godotengine.org/.

20. *Unity at 10: For better—or worse—game development has never been easier*. 2016. Dostupné také z: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

21. *Apple Design Award winners announced*. 2006. Dostupné také z: https://arstechnica.com/gadgets/2006/08/4937/.

22. *How Unity3D Became a Game-Development Beast*. 2013. Dostupné také z: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/.

23. *Unity 2.0 game engine now available*. 2007. Dostupné také z: https://www.macworld.com/article/187693/unity-18.html.

24. *Unity 3 brings very expensive dev tools at a very low price*. 2010. Dostupné také z: https://arstechnica.com/information-technology/2010/09/unity-3-brings-very-expensive-dev-tools-at-a-very-low-price.

25. *Unity officially releases its new game engine: Unity 5*. 2015. Dostupné také z: https://www.theverge.com/2015/3/3/8142099/unity-5-engine-release.

26. *Unity 5 Announced With Better Lighting, Better Audio, And "Early" Support For Plugin-Free Browser Games*. 2014. Dostupné také z: https://techcrunch.com/2014/03/18/unity-5-announced-with-early-support-for-plugin-free-browser-games/.

27. *Unity dropping major updates in favour of date-based model*. 2017. Dostupné také z: https://www.gamesindustry.biz/articles/2016-12-14-unity-dropping-major-updates-in-favour-of-date-based-model.

28. *Introduction to URP*. 2019. Dostupné také z: https://learn.unity.com/tutorial/introduction-to-urp.

29. *High Definition Render Pipeline overview*. 2020. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/index.html.

30. *About the Lightweight Render Pipeline*. 2020. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.render-pipelines.lightweight@5.10/manual/index.html.

31. *System requirements for Unity 2020 LTS*. 2020. Dostupné také z: https://docs.unity3d.com/Manual/system-requirements.html.

32. *Hybrid Renderer*. 2020. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.11/manual/index.html.

33. *Introduction to ShaderGraph*. 2019. Dostupné také z: https://learn.unity.com/tutorial/introduction-to-shader-graph.

34. *Unity Scripting*. 2020. Dostupné také z: https://docs.unity3d.com/Manual/ScriptingSection.html.

35. *Unity Execution Order*. 2020. Dostupné také z: https://docs.unity3d.com/Manual/ExecutionOrder.html.

36. *GDF Format* [online]. 2017 [cit. 2021-06-03]. Dostupné z: https://gephi.org/users/supported-graph-formats/gdf-format/.

37. *GEXF Format* [online]. 2017 [cit. 2021-06-03]. Dostupné z: https://gephi.org/gexf/format/.

38. *StreamReader*. Dostupné také z: https://docs.microsoft.com/cs-cz/dotnet/api/system.io.streamreader.

39. *StreamWriter*. Dostupné také z: https://docs.microsoft.com/cs-cz/dotnet/api/system.io.streamwriter.

40. *igraph - The network analysis package*. 2020. Dostupné také z: https://igraph.org/.

41. *DataTable*. Dostupné také z: https://docs.microsoft.com/cs-cz/dotnet/api/system.data.datatable.

42. *What is XML?* Dostupné také z: https://www.w3schools.com/whatis/whatis_xml.asp.

43. *How C Reflection Works With Code Examples*. 2017. Dostupné také z: https://stackify.com/what-is-c-reflection/.

44. *XDocument Třída*. Dostupné také z: https://docs.microsoft.com/cs-cz/dotnet/api/system.xml.linq.xdocument.

45. *Maximizing Your Unity Game's Performance* [online]. 2017 [cit. 2020-12-30]. Dostupné z: https://cgcookie.com/articles/maximizing-your-unity-games-performance.

46. *Unity Draw Call Batching: The Ultimate Guide* [online]. 2020 [cit. 2020-12-30]. Dostupné z: https://thegamedev.guru/unity-performance/draw-call-optimization/.

47. *What is a Job System?* [Online]. 2018 [cit. 2021-01-02]. Dostupné z: https://blogs.unity3d.com/2018/10/22/what-is-a-job-system/.

48. *Entity Component System for Unity: Getting Started* [online]. 2020 [cit. 2021-01-02]. Dostupné z: https://www.raywenderlich.com/7630142-entity-component-system-for-unity-getting-started.

49. *Unity Job System and Burst Compiler: Getting Started* [online]. 2020 [cit. 2021-01-02]. Dostupné z: https://www.raywenderlich.com/7880445-unity-job-system-and-burst-compiler-getting-started.

50. *Data-Oriented Technoloy Stack* [online]. 2019 [cit. 2021-01-02]. Dostupné z: `https://unity.com/dots`.